

(2)

AD-A226 985

IDA DOCUMENT D-754

REUSE IN PRACTICE WORKSHOP SUMMARY

James Baldo, Jr.

April 1990

DTIC
ELECTE
SEP 12 1990
S E D

Prepared for
Strategic Defense Initiative Organization



DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

INSTITUTE FOR DEFENSE ANALYSES

1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

90 08 12 043

IDA Log No. HQ 90-035350

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Document is published in order to make available the material it contains for the use and convenience of interested parties. The material has not necessarily been completely evaluated and analyzed, nor subjected to formal IDA review.

© 1990 Institute for Defense Analyses

The Government of the United States is granted an unlimited license to reproduce this document.

Approved for public release, unlimited distribution. Unclassified.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE April 1990		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Reuse in Practice Workshop Summary			5. FUNDING NUMBERS MDA 903 89 C 0003 T-R2-597.2	
6. AUTHOR(S) James Baldo, Jr.				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard Street Alexandria, VA 22311-1772			8. PERFORMING ORGANIZATION REPORT NUMBER IDA Document D-754	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Organization (SDIO) SDIO/ENA The Pentagon, Room 1E149 Washington, DC 20301-7100			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Public release/unlimited distribution.			12b. DISTRIBUTION CODE 2A	
13. ABSTRACT (Maximum 200 words) IDA Document D-754 summarizes the Reuse in Practice Workshop which was held at the Software Engineering Institute. The objective of this workshop was to assess the current state of the practice of software reuse and provide recommendations to the research and user communities to enhance software reuse. The workshop focused on four areas of software reuse: domain analysis, implementation, environments, and management. Position papers from several of the attendees are included as part of the document.				
14. SUBJECT TERMS software reuse; domain analysis; implementation; environments; black box reuse; domain modeling; robust; reusable components; traceability.			15. NUMBER OF PAGES 224	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR	

IDA DOCUMENT D-754

REUSE IN PRACTICE WORKSHOP SUMMARY

James Baldo, Jr.



April 1990

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003

Task T-R2-597.2

Preface

IDA Document D-754, *Reuse in Practice Workshop Summary*, was prepared for the Strategic Defense Initiative Organization (SDIO) in response to tasking contained in IDA Task Order T-R2-597.2 under contract MDA 903-89-C-0003.

This document is the result of a workshop held in Pittsburgh, PA, 11-13 July 1989. The objective of the workshop was to assess the state of the practice of software reuse. Software reuse practice was evaluated in the following areas: domain analysis, implementation, software environment support, and management. However, due to the workshop program committee's objective to produce results that were detailed enough to have impact, other software reuse topics, for example software reuse libraries, were not included.

The document contains an executive summary written by the author, summaries of working groups written by the chairpersons and rapporteurs, and a collection of position papers submitted by the workshop attendees. It should be noted that the summaries and position papers do not necessarily reflect the views of Institute for Defense Analyses (IDA), Software Engineering Institute (SEI), Association for Computing Machinery (ACM), or the Strategic Defense Initiative Organization (SDIO).

The author would like to thank Sylvia Reynolds for her editorial advice and assistance.

Acknowledgments

The author wishes to thank the many people who assisted in organizing and running this workshop and the sponsoring organizations, IDA, SEI, SDIO and the ACM. Without them this workshop would not have been possible or successful.

A great amount of credit for this workshop goes to my co-chair Chris Braun. Her keen insight into the working group selections and her heavy effort through e-mail messages, phone calls, position paper reviews, and meetings, were key to the successful organization of the workshop.

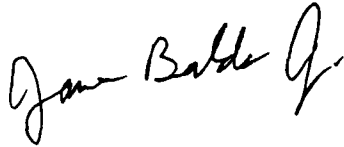
The SEI is to be commended for their efforts in overcoming a disaster the weekend before the workshop. Power to the SEI building, the originally intended site of the workshop, was lost for an entire week due to the flooding of a basement that housed their electrical generators. Sholom Cohen of SEI managed to obtain another facility and make arrangements in record time for necessities such as food and overhead projectors. Sholom handled all the local arrangements for the workshop in superb fashion.

The keynote speakers, Will Tracz and Ted Biggerstaff, set the momentum of the workshop through keynote addresses that both motivated and framed the discussion and generation of ideas. Both Will and Ted assisted Chris and I by providing helpful suggestions with respect to the organization of the workshop.

Also, many thanks to the working group chairmen and rapporteurs. LtCol. Charles Lillie (chairman) and Terry Bollinger (rapporteur) of the Management Working Group, Dan Hocking (chairman) and William Novak (rapporteur) of the Environment Working Group, Ted Biggerstaff (chairman) and Kyo Kang (rapporteur) of the Domain Analysis Working Group, and Will Tracz (chairman) and Stephen Edwards (rapporteur) of the Implementation Working Group.

And most importantly thanks to the attendees who devoted three days to discussing and generating information on software reuse.

Program Co-Chair

A handwritten signature in cursive script, reading "James Baldo, Jr.", written in dark ink.

James Baldo, Jr.

Executive Summary

The Reuse in Practice Workshop was held at the Software Engineering Institute, in Pittsburgh, PA, 11-13 July 1989. The objective of this workshop was to assess the current state of the practice of software reuse and provide recommendations to the research and user communities to enhance software reuse. The workshop focused on four areas of software reuse: domain analysis, implementation, environments, and management.

Forty-eight people attended the workshop: twenty-two from the research community; twelve from government; and fourteen from industry (see Appendix C for names and addresses of attendees). The research community consisted of universities and Federally Funded Research and Development Centers (FFRDCs). The federal government was represented primarily by the Department of Defense (DoD) and other industry representatives were in attendance.

Will Tracz and Ted Biggerstaff started the workshop with keynote addresses. The central theme of Will Tracz's keynote address was, "Where does reuse start?" he defined software reuse as:

...the process of reusing software that was designed to be reused. Software reuse is distinct from software salvaging, that is, reusing software that was not designed to be reused, furthermore, software reuse is distinct from carrying over code, that is, reusing code from one version of an application to another.

Will structured his keynote address by discussing the three P's of software reuse: Product, or what we reuse; Process, or when do we apply reuse; and Personnel, who make reuse happen. Will elaborated on each of these in his keynote address with the goal of identifying issues surrounding software reuse and relating this information to the theme question, "Where does reuse start?" The complete text of the keynote address is provided in Appendix B.

Ted Biggerstaff's keynote address focused on domain analysis. Ted's central theme was to describe domain analysis and domain modelling and their relationship to software reuse. Domain analysis is used for the following:

- a. Black box reuse;
- b. Reuse with modification;
- c. Harvesting reusable components;
- d. Aiding understanding;
- e. Capturing technological methods (intellectual property); and
- f. Aiding training.

Ted also emphasized that an important attribute of domain analysis is to provide human understanding. He explained domain modelling based on an example of a window manager for a computer workstation. Ted finished his talk by describing current research activities at the Microelectronics and Computer Technology Corporation (MCC) in domain analysis.

The workshop was composed of four working groups: domain analysis, implementation, environment, and management. Each group identified issues in their area based on current state of the practice for software reuse and provided a potential approach to the issues or provided recommendations that the research community should address.

The Domain Analysis Working Group identified the need for a general domain analysis model that would support a foundation and context for the practice of software reuse. The group generated a domain analysis model called the Pittsburgh Workshop Model of Domain Analysis. The model divides a domain into three parts: problem space, solution space, and mapping between the two. Basically the model represents a problem space in terms of product features (e.g., a relational data base management system), underlying principles (e.g., the relational algebra), and relationships between product features and principles (mapping between relational data bases and hierarchical data bases). The solution space is described in terms of design criteria, design

alternatives for components, architectural information about a specific target system, constraints among the architectural components, and the implementation components with all of their associated architectural commitments.

The Implementation Working Group was concerned with producing reusable software components that are robust, reliable, understandable, and easy to use, admittedly a difficult goal. In an attempt to consider this goal, the group concentrated on defining and refining terms and processes associated with using domain analysis information. In addition, the Group focused on the generation of parameterized modules that could be reused with a high degree of confidence. The group defined two models as a basis for building reusable software components: a Process Model for creating parameterized components and a Conceptual Model. The Process Model describes a sequence of steps, using domain analysis information to derive parameterized software components base on the Conceptual Model.

The Conceptual Model for reusable software components is based on three ideas:

- Concept - what abstraction the component embodies;
- Content - how that abstraction is implemented; and
- Context - the software environment necessary for the component to be meaningful.

A simple mapping of these ideas to Ada may help assimilate the model: the concept might become a generic specification, each separate content might become a different body for that specification, and contextual decisions might be represented as the formal generic parameters in the specification.

The Environment Working Group examined the software engineering process to identify changes that would abet reuse and map those changes to current software engineering environments. The group identified the following software engineering approaches that need to be integrated into the environment:

- a. Functional rapid prototyping with reusable components;
- b. Process for identifying potential candidates for reuse;

- c. Methods to evaluate software components for reuse; and
- d. Capabilities to physically retrieve software components from a library.

The group noted that a reuse environment must support automated traceability of a component through the requirements to the executable object. Traceability is important for a user understanding of the component's design and implementation, since it captures the context and the constraints of the development process. This understanding assists a user of the component in reusing it on another application.

The Management Working Group agreed that existing reuse technology is being inhibited from practice based on current software management and policy structures throughout the industry. The reason for this problem is based on a lack of understanding by management of "how" and "why" software reuse could benefit the software development process.

To facilitate and encourage software reuse in the short-term, the group made several recommendations, such as demonstrating reuse technology on a project, providing incentives for reuse on contracts, and education and training. For the long-term, the group recommended better technical and administrative support for reuse in the lifecycle work products other than code, such as designs, specification, and test data. They also recommended updating or developing regulations and standards to address reuse explicitly.

In summary, the workshop groups provided the following key observations and products:

- a. A domain analysis model, which attempts to address the domain problem space, solution space, and a mapping between the two;
- b. A proposed conceptual model for software components and a process model for developing and using reusable software components, based on the conceptual model, was defined;
- c. Extensive analysis of the software engineering environment with respect to support of software reuse; and
- d. Identification of management issues inhibiting software reuse and recommendations to increase software reuse in the lifecycle work products.

Table of Contents

1. SUMMARY OF DOMAIN ANALYSIS WORKING GROUP	1
1.1 INTRODUCTION	1
1.2 A DOMAIN MODEL	1
1.3 ISSUES AND NEEDS	6
1.4 RECOMMENDATIONS AND RATIONALE	8
2. SUMMARY OF IMPLEMENTATION WORKING GROUP	10
2.1 INTRODUCTION AND SCOPE	10
2.2 RESULTS	10
2.3 CONCEPTUAL MODEL FOR SOFTWARE COMPONENTS	11
2.4 EXAMPLE	13
2.5 PARAMETERIZATION PROCESS MODEL	14
2.6 ISSUES/NEEDS	16
2.7 RECOMMENDATIONS	17
2.8 RECOMMENDATION RATIONAL	17
2.9 STATE OF THE PRACTICE	18
3. SUMMARY OF ENVIRONMENT WORKING GROUP	20
3.1 INTRODUCTION	20
3.2 STATE OF THE PRACTICE	20
3.3 ISSUES AND NEEDS	21
3.4 ROLES AND RESPONSIBILITIES	22
3.5 ENVIRONMENT TOOL CATEGORIES	25
3.6 RECOMMENDATIONS	28
4. SUMMARY OF MANAGEMENT WORKING GROUP	29
4.1 INTRODUCTION AND SCOPE	29
4.2 ISSUES AND NEEDS	29
4.3 RECOMMENDATIONS - NEAR-TERM	30
4.4 RECOMMENDATIONS - LONG-TERM	31
4.5 RATIONALE FOR RECOMMENDATIONS	32
APPENDIX A WORKSHOP PROGRAM AND GROUP COMMITTEES	33
APPENDIX B WHERE DOES REUSE START	36
APPENDIX C WORKSHOP ATTENDEES	47
APPENDIX D WORKSHO POSITION PAPERS	57

**Kyo Kang
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890**

1. SUMMARY OF DOMAIN ANALYSIS WORKING GROUP

1.1 INTRODUCTION

The potential ubiquity, incomplete understanding, difficulties, and cost of software reuse indicate the need for a foundation and context for the practice of software reuse. While there are several approaches to reuse, each of them can be supported by domain analysis, which can make a contribution to the needed foundation and context. Therefore the Domain Analysis Working Group focused on the formulation of a general domain analysis model to provide guidance in realizing that support.

1.2 A DOMAIN MODEL

The main result of the Domain Analysis Working Group was a model of domain analysis called the Pittsburgh Workshop Model of Domain Analysis (or PWMDA). The central idea of this model is that a given domain (e.g., the domain of data base management systems) contains three parts:

- a problem space,
- a solution space, and
- a mapping between the two.

The problem space is a network of: 1) target product features (e.g., a relational DBMS); 2) the underlying principles (e.g., the relational algebra); 3) analogies (e.g., tables as an analog of a relational DB); and 4) relationships (e.g., the relationship or mapping between relational DBs and hierarchical DBs).

The solution space contains five general classes of elements: 1) design issues or criteria that represent key questions distinguishing between classes of designs or architectures (e.g., "What is the organizing principle of a DBMS system?"); 2) the specific decision alternatives associated with each issue (e.g., for the question in [1], the alternatives are relational, hierarchical, network, or object-oriented); 3) an architectural component containing the knowledge currently specified about the target system (e.g., it is relational); 4) constraints among the architectural components (e.g., if a network DBMS is chosen, some form of indexing support is required); and 5) the implementation components with all of their associated architectural commitments (e.g., an implementation package for b-trees, used to implement the indexes of the DBMS).

The architectural component may vary in its complexity from a simple assertion about the design (e.g., "DBMS is relationally organized") for highly abstract components, to a detailed pdl specification for highly concrete components. Each component is thought to have a rich substructure that includes architectural patterns, specs, test data and constraints. The constraints which are not drawn explicitly in the network of Figure 1, establish design dependencies among architectural components such that a design decision at one point may influence or restrict a design decision at some far removed point in the solution space network.

The solution space network is composed of partially independent islands or clusters. Each such island may be attended to in a (mostly) arbitrary order during the design process, restricted only by the ordering dependencies imposed by the constraint relationships between decisions scattered among the various islands. (See Figure2.) Within an island, the decisions may be ordered by the structure of the issue-decision-architecture relations.

Figure 3 illustrates the nature of a domain island at the lowest level of detail within the solution domain. In this figure, the architectural components near the top of the island contain abstract algorithms while those near the bottom are the actual code components.

The mapping is the relationship between the features and principles in the problem domain with the issues, decisions and architectural components in the solution domain.

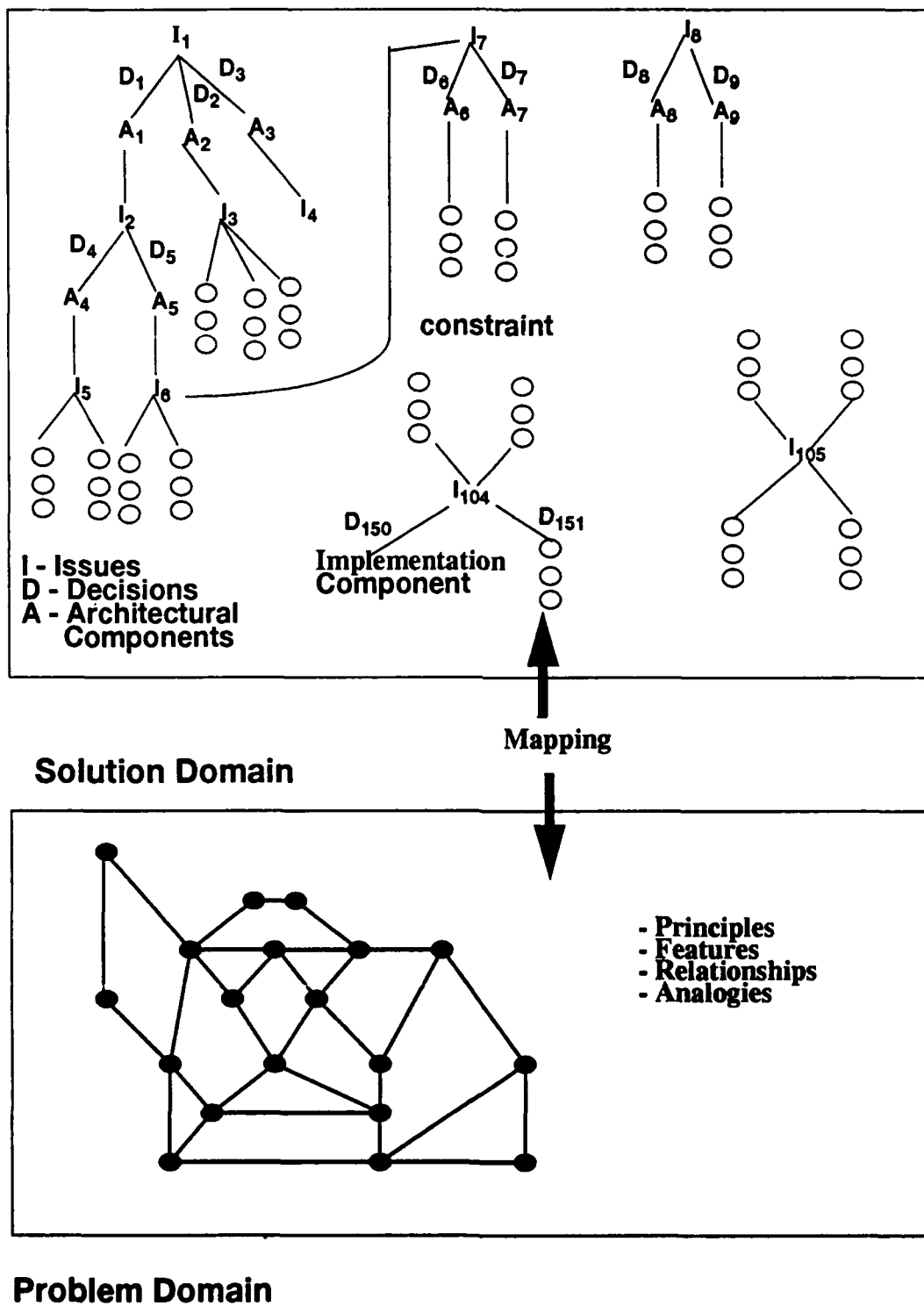


Figure 1: Pittsburgh Workshop Model of Domain Analysis

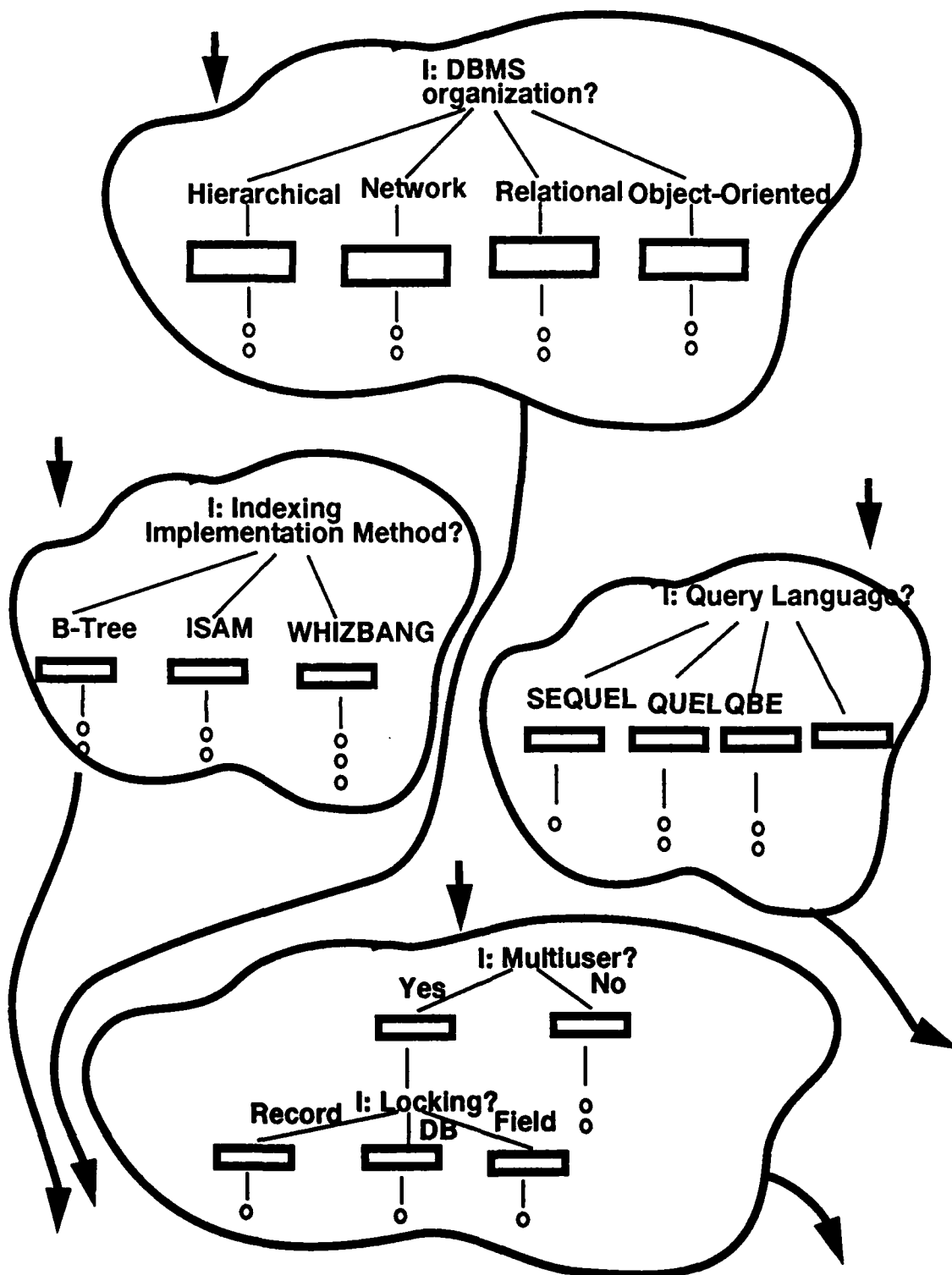


Figure 2: Solution Domain Islands (High Level) for DBMS Domain

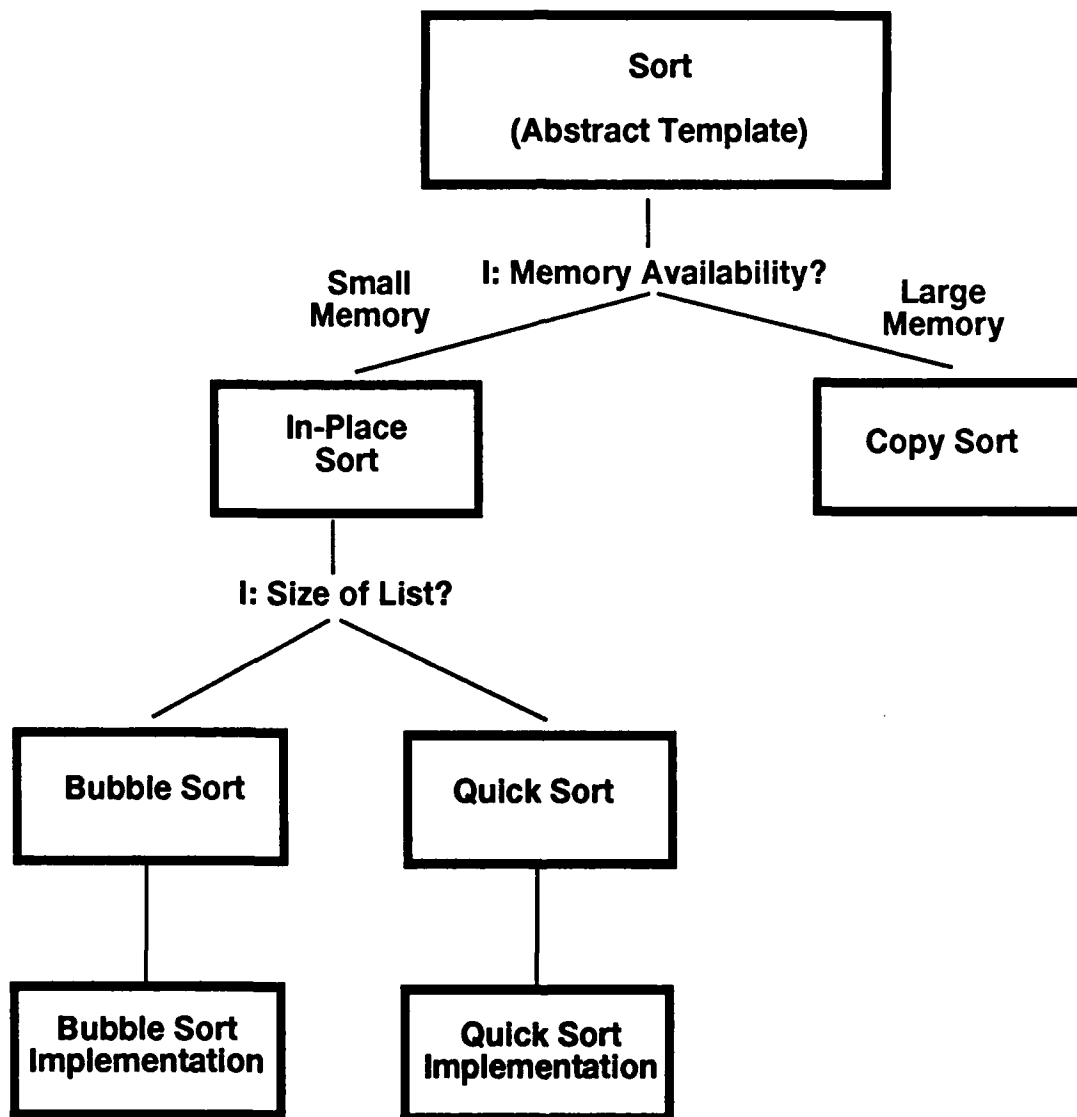


Figure 3: Lower Level Island In solution Domain of DBMS

1.3 ISSUES AND NEEDS

The working group defined a model of domain analysis (i.e., PWMDA). Some of the areas that need further investigations are:

- The model represents a problem space in terms of product features, underlying principles, and relationships between product features and principles. One of the issues has to do with generality and applicability of this model. There could be domains where a different model (e.g., entity-relationship model) might be more appropriate to represent the problem space than the proposed model. Although the group felt positive about the model, all agreed that it needs to be completed and applied to a number of domains to validate the applicability.
- The proposed model is far from complete. The types of relationships must be identified (for the "elements" in the problem space) and representations of the "elements" (i.e., issues, decisions, and architectures) in the solution space must be detailed. Although the model provides a good starting point, a substantial amount of effort still needs to be invested to complete it.
- Another area to investigate has to do with specification and documentation of the "elements" in a domain model. For example, different specification may be needed for code templates than code components that are used without instantiation. The context in which a template can be used must be specified and documented. For each element type of the domain model, what and how to specify and document the elements need to be investigated.

Domain analysis encompasses a number of systems and, therefore, can go across project boundaries in a corporate environment. Also, the results of an analysis must be incorporated into system developments at different projects. This raises a number of project management/organizational issues. Some of the issues discussed at the workshop are:

- Domain analysis is believed to require a very large up-front investment.¹ This cost is amortized over a number of system developments. The producer of a domain model makes a large investment and the users of the model benefit. Often, the producer and the users of a domain model do not belong to the same organizational entity (e.g., project), and the problems that can occur in this case are (1) how to transfer cost to the user organizations, or how to give incentives to make the investment and produce domain models, and (2) how to manage and maintain the library containing domain models and artifacts.
- Various infra-structures (e.g., an organization specialized in domain analysis) were discussed but no specific recommendation was made. The group felt that an infra-structure must be created considering such factors as the corporate structure and the culture.
- In order to effectively utilize domain models in the software development, a systematic approach should be employed during the development. However, the existing life-cycle models (e.g., DoD 2167A) do not address domain analysis and application of domain analysis results in the development. Domain analysis, library management,

1. The experience by the Domain Specific Software Architecture (DSSA) Project at the SEI does not support this view. They could develop a reusable software architecture within the original development schedule without any extra cost. Their approach is somewhat different from the one proposed by the working group in that they extracted a reusable architecture from *one* system. They expect to evolve this architecture through applications.

and domain model application activities should be incorporated into the development life-cycle.

- Another concern raised was that many of existing policies and development culture mitigate against the development of reusable assets and the population of library. Budget constraints and time pressures often prevent one from studying a problem with a broader perspective.

1.4 RECOMMENDATIONS AND RATIONALE

The Domain Analysis Working Group made a set of recommendations to address some of the issues raised at the workshop. Each recommendation is described below along with the rationale behind the recommendation.

- *DoD should fund prototype creation to explore the domain analysis model.*

The domain analysis model proposed by the working group is far from complete. A large investment is expected to be needed to complete the model, develop a prototype, apply the model to a realistic domain, and validate the concept. Government, especially DoD, is expected to make a substantial gain from this approach, and we recommend that DoD fund prototype creation to explore the concept.

- *For each domain analysis, state:*

objective or purpose,

deliverables (concrete),

clear termination or completion criteria, and

scope (boundary).

Since domains (e.g., horizontal/vertical domains) are inter-related, where (breadth) and when (depth) to stop a domain analysis are not always clear. We strongly recommend that the purpose, deliverables, completion criteria, and scope be defined prior to the analysis.

- *Plan for maintenance of the domain model*

A domain model changes as we gain more experience with the domain and with the model. The knowledge gained from the application of the model must be incorporated into the model and maintenance of the model should be an on-going activity. A maintenance plan with allocation of necessary resources should be developed for each domain model.

- *Domain analysis should explicitly define the mapping between the problem space and the solution space.*

The purpose of domain analysis is to analyze how the problems in a problem space are addressed by the solutions in a solution space. Mapping between the problem space and the solution space should be explicitly defined in the model.

**Will Tracz
IBM SID MD 0210
Owego, NY 13827**

2. SUMMARY OF IMPLEMENTATION WORKING GROUP

2.1 INTRODUCTION AND SCOPE

The implementation working group consisted of three industry, two academic, one government, and five research consortium (SEI) members. The major focus of the group was to define and refine terms and processes associated with using domain analysis information for the generation of parameterized modules that could be reused with a degree of certainty as to their validity and effectiveness.

A conceptual model for software components was proposed and a process model for developing and using reusable software components, based on the conceptual model, was defined.

The group also focused on potential extensions to Ada that would support such a software reuse paradigm. These included extending the type model and parameterization mechanisms. Finally, the group discussed the role inheritance plays in software reuse noting the differences between type inheritance and code inheritance.

This summary presents a rough outline of the formal models developed by the working group. A full technical paper refining the formal models is under development by several members of the working group. A copy of this paper is available by contacting the working group leader at the address above.

2.2 RESULTS

The working group strived to define a formal basis for the development and application of reusable software components. To this end, a conceptual model for reusable components was discussed and a process model for creating parameterized components, based on the conceptual

model, was defined. It was assumed that an application domain or business area could be sufficiently defined to warrant analysis, design, and development of software consisting of parameterized modules/objects/components that allowed the rapid creation of new systems (through reuse) within that problem space.

2.3 CONCEPTUAL MODEL FOR SOFTWARE COMPONENTS

The conceptual model for reusable software components was an outgrowth of the Concept/Context model initially proposed by Tracz in his dissertation work at Stanford. The model, referred to as the 3C model (Concept/Context/Content) is based on defining three facets of a software component:

- (1) The "concept" behind a reusable software component is an abstract canonical description of "what" a component does. Concepts are identified through requirement or domain analysis as providing desired functionality for some aspect of a system. A concept is realized by an interface specification and an (optionally formal) description of the semantics (as a minimum, the pre- and post-conditions) associated with each operation. An Ada package specification with its behavioral semantics described in Anna is an example of a reusable software concept.
- (2) The "content" of a reusable software component is an implementation of a concept, or "how" a component does "what" it is supposed to do. The basic premise is that each reusable software component can have several implementations that obey the semantics of its concept. The collection of (28) stack packages found in Grady Booch's components is an example of a family of implementations for the same concept (a stack).
- (3) The "context" of a reusable software component is 1) the environment that the concept is defined in ("conceptual context"), and 2) the environment it is imple-

mented under ("contentual context"). It is very important to distinguish between these two types of contexts because different language mechanisms (inheritance and genericity) apply differently to each. Furthermore, these two contexts clearly distinguish between type inheritance and code inheritance.

One can use type inheritance to describe the concept of a software component in terms of the operations and types found in another software component (what we are calling its concept). In other words, by using inheritance one can describe a new concept in the context of an existing concept. At the conceptual level then, the new concept "is a" specialization (subtype, or subclass) of the parent concept. Aggregation of concepts is accomplished through multiple inheritance. Parameterization or genericity also applies to concepts, but its use is normally associated with passing data or furnishing contextual information such as the type of data or data structure being manipulated (operational context). In the 3C model, parameterization and inheritance play different roles at the conceptual level.

Code inheritance may or may not be used in an implementation. One need not observe conceptual relationships to access operations that may prove useful for the implementation of a software component. There are two separate contexts that apply to an implementation of a software component: a visible context, one that the user can manipulate (operational context), and a hidden context, one the developer has chosen to use in the actual implementation (implementation context).

Interestingly enough, both the operational context and implementation context present opportunities for variations. A software component's operational context is established by the user when, at instantiation or run-time, actual parameters are supplied for formal generic parameters. The implementation context is usually not visible to the end-user of a software component and established at build time. The component developer imports a specific software component or module whose operations are invoked by that particular implementation. But, given the environment defined by the 3C model, it is possible that several implementations could exist that satisfy

the semantic and syntactic properties of the module or component being imported or inherited by the developer. Furthermore there is no reason why certain aspects of the implementation context cannot be tied directly to the operational context. For example, if the user specifies that "fast, bounded" stack of integers is desired, then the stack package's implementation, might import a list package that has been implemented as an array, rather than a linked list.

One should note that while it is often the case that the concept and content of a component share the same context, the context of an implementation often subsumes that of the concept and extends it with performance trade-offs, hardware platform, operating system, algorithmic, or language dependent contextual information. An example of a parametric conceptual context is the type of element to be stored in a generic stack package (an instantiation parameter). An example of a semantic conceptual context is describing a stack in terms of a deque where certain operations are renamed and others are hidden. An example of an implementation's operational context is a conditional compilation variable that selects between UNIX and DOS operating system calls. An example of a component's implementation context is the importation of a list package (which may have several implementations).

2.4 EXAMPLE

The example used for discussion by the working group was the concept of sorting. The sorting concept can not be described without a context². The context used to describe the concept of sorting includes a list of elements that have a partial order on them (a characteristic of the elements in the list). Therefore the context associated with sorting concept is the

2. Note: All concepts and contents have a context! In fact one of the most common problems programmers face with trying to reuse previously written software is determining the assumptions made by the original developer. These assumptions often encompass the contextual information that is buried in the interface or implementation and point out the need to separate the context from the concept and content of a reusable software component!

data and data structure being sorted. This data structure must have certain properties associated with it, that is, its context can, in turn, be described in terms of properties of its elements - that a linear order relationship is defined on them. This is an example of a concept (sorting) whose context (lists) is itself a concept that has a context of the linear order relationship on its elements. Note also, the linear order relationship can be satisfied in many ways (e.g., less than, greater than, is a member of). These are all examples of the conceptual context of the sorting software component.

Focusing now on contentual context, there can be several implementations of lists (e.g., linked list, arrays, or files), therefore the content or implementation associated with the concept can take any number of forms based on different contexts. Similarly, there exists several sorting algorithms, each perhaps more suited for different implementations and attributes of the data (e.g., nearly sorted data), each having different run-time performance and resource utilization characteristics.

The selection of an implementation, or the content of the concept is determined by trade-offs in context. Clearly, knowing the characteristics of the type of data structure being manipulated will lead to more efficient implementations. This can result in the population of a reuse library with several efficient implementations of the same (parameterized) concept, each tailored to a particular context. At design time, a programmer could identify the concept and define the context it is being manipulated under based on requirements or operating constraints. At implementation time, the programmer could instantiate an implementation of the concept with the conceptual contextual information plus any other contentual contextual information necessary.

2.5 PARAMETERIZATION PROCESS MODEL

The parameterization process model describes the sequence of steps for using domain analysis information to derive parameterized software components based on the 3C conceptual model. One should recognize that domain analysis information can be gathered top down or bottom up. Top-down domain analysis starts with an entity-relationship model of an application

domain and determines the components associated with it. Bottom-up domain analysis is based on analyzing several existing systems in an application domain.

In general, a domain analyst 1) identifies a concept 2) determines if variations exist, 3) factors out the commonality and 4) provides selection parameters that specify the context. Alternatively, a concept can be generalized over a range of contextual values. It becomes an economic issue and implementation trade-off as to how many implementations are associated with each concept. Clearly, one general purpose implementation might lead to certain inefficiencies, therefore, several implementations, separated and selectable by context is often desired. A concept may have several implementations, each spanning a subset of the possible solution space bounded by the contextual information associated with the concept.

The process of developing reusable software components based on the 3C conceptual model (the process of separating concept from context, content from concept, and context from content) may be described as follows:

(1) "Separating Concept from Content "

- Analyze an application domain. Recognize commonality of some functionality within an application domain.
- Use commonality to define a concept.
- Isolate differences or variations of the functionality.
- Isolate differences or variations in possible implementations of the functionality.
- Record implementation issues for later use.
- Define an interface to the concept in the form of an Abstract Data Type.
- Define the semantics of the concept as pre- and post- conditions (as a minimum).

(2) "Separating Concept from Context"

- Use difference to define a context of the concept.
- Given a concept and its context, iterate and generalize the concept by expanding the context.

- Given a concept and its context, iterate and refine the concepts and its context. Continue until the concept and its context are defined in terms of basic concepts (hopefully a set of which are in the reuse library).
Note: Inheritance type hierarchies are useful in expressing certain concepts in terms of related concepts.
- Refine the interface to the concept, if necessary, taking into account contextual information.
- Refine the semantics of the concept, if necessary.

(3) "Separating Context from Content"

- Define the context of the content. Determine the implementation variations and dependencies (e.g., operating system, hardware, or compiler dependencies).
- Define a context of interest for the concept.
- Define a context of interest for the content.
- Implement variations of the concept according to trade-offs on performance and resources with respect to the context of interest.
- Verify that the content (each implementation) matches the concept.

2.6 ISSUES/NEEDS

The reuse issues addressed by the working group focused on populating a reuse library with robust and reliable reusable components that are easy to locate, understand, and use. The need for sufficiently adaptable, portable and re-configurable software was felt to be addressed by using parameterization to separate out the aspects of software that make it not reusable (implementation dependencies, i.e., contextual information embedded in the implementation). The number and types of parameters, as well as efficiency issues were also recognized as playing crucial roles in determining the "usability" of the reusable software. Application generators, the modularization of parameters (parameterized parameters), and expert system assistance (e.g., AMPEE in CAMP) were cited as possible approaches for controlling complexity.

2.7 RECOMMENDATIONS

The working group suggested the following as recommendations for future activities in the field of software reuse:

- (1) Ada 9x should consider adding package and procedure types to support the development of reusable software in Ada.
- (2) The Concept/Context/Content conceptual model should be refined and discussed in the programming community in the large.
- (3) Application domains should be selected for domain analysis and the generation of parameterized modules developed and documented consistent with the parameterization process model.

2.8 RECOMMENDATION RATIONAL

Motivation for the recommendations can best be summarized in the words of three of the individuals who attended the workshop.

- (1) "There exists no good conceptual basis to apply to software reuse." -- Bruce Barnes, NSF. The 3C model provides a good conceptual model for the development of reusable software with a formal foundation in type and category theory.
- (2) "Understanding depends on expectations based on familiarity with previous implementations." -- Mary Shaw, SEI. One of the failures of software reuse is that the expectations of the user of the reusable software do not meet the expectations of the designer of the reusable software. By explicitly defining the context of a reusable software component at the concept and content level, and formally defining its domain of applicability, the user can better select and adapt the component for reuse.
- (3) "Domain analysis is building up a conceptual framework, informal ideas and rela-

tions; the formalization of common concepts". -- Ted Biggerstaff, MCC. Domain analysis is the key to identifying and specifying reusable software modules (concepts). The parameterization process model is an approach for organizing and representing this knowledge based on clearly defined relationships stated in the 3C conceptual model.

Finally, it was the general consensus of the group that as the number of parameters in a module increases, the ease of use decreases. Therefore the introduction of package types as Ada generic formal parameters is desirable as an approach to organizing parameters.

2.9 STATE OF THE PRACTICE

The working group observed a varying degree of reuse technology being incorporated by industry. This broad spectrum of reuse activity can be summarized as follows:

- (1) "No planned reuse."

Software reuse is done on an informal, ad hoc³ basis by salvaging software from previous projects.

- (2) "Pilot projects."

A reuse pilot project is underway that includes studying reuse literature, generating guidelines, and developing a small set of components for reuse.

- (3) "Informal Reuse."

Software is identified by projects or departments as being reusable. Informal guidelines are set up to provide a minimal degree of documentation and testing for entry into the reuse library.

- (4) "Corporate Support."

Upper management has made a commitment to applying software reuse by providing resources and incentives to develop and maintain reusable software repositories that comply with existing reuse guidelines. Reuse and deposition quotas have been set on new

3. Odd Hack is perhaps a more appropriate term.

projects.

So far, only a handful of companies have made significant commitments to reuse (e.g., IBM and GTE). Most other companies are either just starting pilot projects to evaluate reuse technology, or are evaluating results from them. There have been some less than successful projects in some companies, partially due to lack of critical mass and discipline, underestimating the cost of reuse, overemphasizing the creation of reuse tools, or lack of understanding of the technical issues associated with developing reusable software.

William Novak
General Electric
Resident Affiliate
Software Engineering Institute
Software Methods
5000 Forbes Avenue
Pittsburgh, PA 15213

3. SUMMARY OF ENVIRONMENT WORKING GROUP

3.1 INTRODUCTION

The primary objective of a study of support environments for software reuse is to identify the *changes* in the software engineering process required to support software reuse, and the resulting changes required in the software development and maintenance *environment*. This objective arises from the fact that it is the software engineering process which defines the activities to be supported by an environment and set of tools. Since this objective is too broad to satisfy in the course of a single workshop the Environment Working Group tried to lay the groundwork for approaching the problem.

3.2 STATE OF THE PRACTICE

A very brief synopsis of some aspects of the current state of the practice in software reuse is summarized as follows:

- (Re)use of utility routines is common (i.e., math, window, menus, sorts, etc.)
- Stand-alone systems are (re)used often (i.e., compilers, databases, etc.)
- Reusable software is concerned with planned reuse of components⁴ different from sta-

4. The term *component* was selected as the standard term by the working group, rather than others such as *resource*, *asset*, *artifact*, *part*, *element*, or *module*. 'Component' has two senses: an all-inclusive meaning throughout the life-cycle, and a limited meaning as being only executable code; the former sense is the one intended. The connotations associated with the other terms were either pejorative or unclear.

ble (e.g.math) routines and less than stand-alone systems.

- Current use of reusable components is local rather than remote/distributed.
- Code reuse is the only type practiced (as opposed to other life-cycle components)
- *Ad hoc* reuse is more common than engineered reuse.

This overview of current practice provides a context in which to discuss the development of reuse environments.

3.3 ISSUES AND NEEDS

In order to address the issue of how best to support developers using reusable software, the process needs were identified. Some of the most significant changes/additions to the software engineering process which result from software reuse are:

In order to address the issue of how best to support developers using reusable software, the process needs were identified. Some of the most significant changes/additions to the software engineering process which result from software reuse are:

- *Identification of potential candidates for reuse:* This step will become an intrinsic part of many phases of the software life-cycle, since many life-cycle deliverables have the potential to be reused.
- *Evaluate and select appropriate components to be reused:* In the same way, at many phases of the life-cycle the set of available reusable software components must be evaluated and selected for incorporation into the system being developed.
- *Retrieve candidate components for inclusion in the new system:* After evaluation and selection the components must be physically obtained for use in the target system.

One of the factors which affects the nature of a software reuse support environment is the

paradigm under which reuse occurs. Different names for this have been published in the literature, but the same divisions often recur:

- *Constructive*: Most common, with systems built using some existing (code)parts, and software assembled using standard connection techniques (e.g., pipes, etc.)
- *Adaptive*: Design components with high-level parameters for both procedure and data (as in object oriented languages)
- *Generative*: Top-down approach that generates tailor-made components from templates based on requirements schematics/rules.

3.4 ROLES AND RESPONSIBILITIES

The software engineering process defines the activities and the roles which an environment must support. The primary roles identified are those of the *producer* of components, the *user* of components, and the *manager* of components (or library administrator). The following are lists of the activities which are performed in the process of reusing software from the points of view of these three roles.

From the point of view of a user of reusable software components there is a well-defined set of steps which begins at the moment that the possibility of reuse exists. The steps are: 1) *Identify* a set of possible components, 2) *Understand* the function and constraints of each component, 3) *Evaluate* each member of the set of components, 4) *Select* the best component for the application, 5) *Retrieve* the component, and 6) *Integrate* the component with the other application elements

Use of Components

- a. Analysis (of requirements)

- (1) Identify initial functionality and performance requirements
 - (2) Identify, understand, evaluate, select, and retrieve reusable components. Negotiate requirements based on reusable components (and adapt)
 - (3) Identify, understand, evaluate, select, and retrieve reusable components for prototype
 - (4) Develop a functional prototype system
 - (5) Trace requirements to design
 - (6) Adapt design to use components
- b. Design (of solution)
- (1) Identify, understand, evaluate, select, and retrieve design components
 - (2) Prototype and compare alternate designs
 - (3) Transform requirements components into design components
- c. Implementation
- (1) Identify, understand, evaluate, select, and retrieve code components
 - (2) Prototype alternative implementations for comparison
 - (3) Trace design elements to requirements and to code
- d. Test
- (1) Use test cases stored with components in the library
- e. Maintenance
- (1) Improve training and understanding by maintainers through the domain model, better available information on the system
 - (2) Software problems handled through the reusable library system and propagated to

other users

In general, component *producers* provide reusable components, documentation, test plans and cases, initial classification information, and fixes and enhancements for the components.

Production of Components

- a. Identification (through domain analysis, application development, and harvesting from existing inventory)
- b. Engineering for reuse (includes documentation)
 - (1) Using guidelines for development and implementation
 - (2) Metrics to determine component's performance and other characteristics
 - (3) Developing components from *scratch* or *harvested* from existing software
 - (4) *Specific* to one application or *generic* across many library paradigm (*Constructive, Adaptive, Generative*)

Validation and Verification

- a. Include review/participation by the domain expert
- b. Testing for context sensitivity

Submission to the Library

- a. Producer certification (may also be done by library)
- b. Standards for design methodology, documentation, coding, etc.

Rapid Prototyping

(The following responsibilities and associated activities fall to the component *manager*, or library administrator.)

Management of Components

- a. Distribution
- b. Training in library use

- c. Configuration management of library
- d. Registration
- e. Classification/organization schemes for the library contents
- f. Cataloguing of the library components
- g. Reviewing the submissions to the library
- h. Testing/Certification of library submissions
- i. Security and analysis of the library contents for viruses and tampering
- j. Safety/Backups/Integrity of the physical library data
- k. History/Metrics/Accounting of library usage
- l. Results of domain analysis

3.5 ENVIRONMENT TOOL CATEGORIES

A set of broad abstract categories was defined to encompass a range of reuse environment support tools and methods. These top-level categories are:

- a. Library procedures
- b. Analysis methods
- c. Guidelines, standards, and policies for both component *production* and *evaluation*
- d. Cataloguing methods
- e. Search mechanisms
- f. Retrieval mechanisms
- g. Physical storage

In defining the activities and tool categories one recurring idea was the importance of

being able to provide the user of a reuse environment with automated traceability from an executable component to the design, requirements, and other information associated with that component. One of the major reasons for this is the ability to understand the context and constraints which influenced the development of the component so that these decisions and rationales may be re-examined in new circumstances when a component is reused.⁵

The following environment tool categories are further subdivided to detail some of the specific support which would be required for a reusable software library system based on a constructive paradigm. These categories are not directly cross-referenced with the roles defined earlier due to the large degree of overlap (i.e., analysis methods are used by producers in validating components, by library managers in accepting components, and by users in testing components prior to use in application systems). Rather, they simply address environment support for 1) system/component production tools, 2) library tools, 3) domain analysis tools, and 4) management tools.

System/Component Production Tools

- a. Component construction/adaptation/generation
- b. Component production "guidelines/standards/policies"
- c. Requirements tools
- d. Design tools
- e. Analysis methods and tools
- f. Traceability tools
- g. Expert systems

Library Tools

- a. User Interface

5. See the results from the Domain Analysis Working Group for further details in this regard.

- (1) Education (on-line help, training, tutorials, etc.)
- (2) View/examine components (to determine appropriateness for an application)
- (3) Traceability between all life-cycle products associated with the component
- (4) Search mechanism (to locate relevant components)

b. Component Submission

- (1) Component evaluation tools (metrics on component suitability)
- (2) Data extraction tools (to partially extract cataloguing information)
- (3) Component data entry support
- (4) Tagging mechanisms
- (5) Security/virus protection

c. Physical data retrieval mechanism

d. Storage (physical security and data integrity)

e. Library Management

- (1) Security/classification
- (2) Cataloguing
- (3) Analysis
- (4) Configuration management
- (5) Component evaluation
- (6) Library procedures
- (7) Expert systems

Domain Analysis Tools

- a. Automated Support for browsing the domain model
- b. Project use of domain analysis results

Management Tools

- a. Project management
- b. Economic/historical collection

3.6 RECOMMENDATIONS

In the course of attempting to define a reuse environment in the context of the software engineering process which it must support, it became apparent that the existing waterfall version of the life-cycle loses too much information during the process (especially in terms of traceability) and that it was inadequate. Clearly, new process models are required to properly handle software reuse, especially the use of the new technologies and paradigms for reuse. Process models designed to support reuse will allow software development environments to be specifically tailored to the process.

Terry Bollinger
CONTEL Technology Center
12015 Lee Jackson Highway
Fairfax, VA 22033-3346

4. SUMMARY OF MANAGEMENT WORKING GROUP

4.1 INTRODUCTION AND SCOPE

The overall goal of the Management Issues Working Group at the SEI / IDA Reuse in Practice Workshop was to develop recommendations for how managers throughout the software industry could put software reuse into practice. The working assumption of the group was that existing methods for reusing software are already well ahead of the actual application of such methods, and that the problem of achieving significant, consistent levels of software reuse throughout the industry is therefore more managerial than technical.

The working group decided that the scope of the issues discussed in the group would address the following five aspects of reuse:

- a. *Problems.* What are the key difficulties to widespread software reuse?
- b. *Benefits.* What are the benefits of reuse, and which of those benefits are the most relevant to managers?
- c. *Incentives.* What incentives can managers use to encourage software reuse in an organization or project?
- d. *Economics.* What are the economics (cost issues) of software reuse?
- e. *Legal.* What are the key legal issues that need to be solved in software reuse?

4.2 ISSUES AND NEEDS

Overall, the issue that seemed to dominate much of the discussion in the management group was the need for a better understanding in the software community of the potential benefits

(and problems) of software reuse. It was felt that having an accurate, widespread understanding of *why* and *how* reuse can be beneficial would assist greatly in promoting its widespread use.

Other issues that the group identified for further discussion included the need for:

- a. Contract incentives.
- b. Education and training.
- c. Better measurement processes.
- d. Clear-cut reuse "success stories."
- e. *Active* work on legal issues.
- f. A multi-dimensional definition of reuse; that is, one which explains reuse in terms of a particular audience's needs.
- g. Upper management and political involvement.

4.3 RECOMMENDATIONS - NEAR-TERM

The near-term recommendations for increasing software reuse in the industry included:

- a. Implementation and full characterization of one or more successful, "full term" reuse projects in which the benefits of reuse can be described clearly and unambiguously.
- b. Explicit mention of reuse in contracts, including in particular the use of incentive fees to specifically promote reuse.

Mr. Stanley Levine gave a specific example of an actual draft contract that specified incentive fees for software reuse in the Army's Advanced Field Artillery Tactical Data Systems (AFATDS) effort.

- c. Education and training. Education of certain key personnel was seen as being a particularly important initial step in promoting reuse. Two key groups are:

- Contracts people
- Program managers

Specific mechanisms for educating these people in reuse could include

- (1) Customized seminars
- (2) Established courses (e.g., Defense Systems Management College, Air Force's Bold Stroke, and others)
- (3) Making managers aware of the non-cost benefits of software reuse. If most managers recognize that reuse can benefit areas such as design and code quality, maintainability, and rapid response to customer needs, they will be more likely to accept it as a problem-solving tool.
- (4) Explicit questions on reuse during program reviews. Asking about reuse (or the lack thereof) during a program review is a simple way to greatly increase the awareness of the potential opportunities for reuse in a project.

4.4 RECOMMENDATIONS - LONG-TERM

The long-term recommendations of the group for increasing software reuse included:

- a. Better technical and administrative support for broad-spectrum reuse — that is, for the reuse of life cycle work products other than code, such as designs, specifications, and test data.
- b. Updating or developing regulations and standards to address reuse explicitly:
 - (1) Develop a reuse manager's guide.
 - (2) Update 2167A to address reuse.
 - (3) Update AFR 800-14 for reuse.

(4) Developing better cost models

4.5 RATIONALE FOR RECOMMENDATIONS

There was a general consensus in the group that the benefits of reuse clearly exist, and that it is those benefits that provide the rationale for the above recommendations. Specific benefits which could accrue from software reuse include:

- a. Reduction of development costs.
- b. Increases in reliability.
- c. Increases in software quality.
- d. Shortening of development schedules.
- e. Reduction of risks
- f. Responsiveness to customer needs.
- g. Ability to build larger, more complex systems.
- h. On-the-job education of developers.
- i. Increased maintainability
- j. Increased security.

It was observed that many or all of these benefits are only potential, and that poorly planned reuse could result just as easily in losses as in gains of the desired qualities. Like other software technologies, reuse is no panacea; only by increasing an overall awareness of when and how to build and use reusable components will reuse become a significant part of the software development process.

APPENDIX A

Workshop Program Committee

James Baldo Jr. - Workshop Co-Chair

Chris Braun - Workshop Co-Chair

Sholom Cohen - Local Arrangements

Working Group Members

Domain Analysis Working Group

Ted Biggerstaff - Chairman

Kyo Kang - Rapporteur

Edward Beaver

Patrick Carroll

Ernesto Guerrieri

Barbara Hignite

Kenneth Lee

Jim Perry

Mary Shaw

Implementation Working Group

Will Tracz - Chairman

Stephen Edwards - Rapporteur

Bruce Barnes

Sholom Cohen

Liesbeth Dusink

John Goodenough

Larry Latour
Spencer Peterson
Chuck Plinta
Ruth Rudolph
Ruth Shapiro

Environment Working Group

Dan Hocking - Chairman
William Novak - Rapporteur
Harley Ham
James Hess
Beverly Kitaoka
Constance Palmer
James Solderitsch
Terry Vogelsong
Paul Wilbur

Management Working Group

Charles Lillie - Chairman
Terry Bollinger - Rapporteur
Gregory Aharonian
Dennis Ahern
Richard Armour
Brian Baker
Richard Fairley
Robert Holibaugh
Harry Joiner
Stanley Levine
James Lund

Rod Moyes

Philip Palatt

Spencer Peterson

APPENDIX B

Where Does Reuse Start

**Will Tracz
IBM SID MD 0210
Owego, NY 13827
OWEGO@IBM.COM or TRACZ@SIERRA.STANFORD.EDU**

Preface

The following is a transcript of the keynote address for the Reuse in Practice Workshop sponsored by IDA, SEI and SIGADA. The workshop was held in Pittsburgh, PA at the Software Engineering Institute, July 11-13th, 1989. The goal of this talk was to establish some common vocabulary and to paint a broad picture of the issues related to software reuse.

Overview

Software reuse is the type of thing some people swear by. It is also the type of thing that some people swear at. Software reuse is a religion, a religion that all of us here today pretty much have accepted and embraced. The goal of this talk is to question the foundation of our faith - to test the depth of our convictions with the hope of shedding new light on our intuitions. I do not claim to have experienced divine intervention. You don't need to take what I say as gospel truth. I believe in what I say, but what you hear may be something different. Again, let me encourage you to disagree - to challenge the position I have taken on the issues I will be presenting. Before I proceed further, I need to qualify software reuse by providing a definition.

Software reuse, to me, is the process of reusing software that was designed to be reused. Software reuse is distinct from software salvaging, that is reusing software that was not designed to be reused. Furthermore, software reuse is distinct from carrying/over code, that is reusing code from one version of an application to another. To summarize, reusable software is software that

was designed to be reused.

The major portion of my talk will focus on examining the rhetorical question, "Where does reuse start?"

Introduction

If I were to ask you, "Where does reuse start?", your reply might be, "What do you mean? That seems like a pretty vague and nebulous question!"

I agree, so I have done a little top/down stepwise refinement and broken the question up to focus on three areas - the three P's of software reuse: product, or what do we reuse, process, or when do we apply reuse, and finally personnel, or who makes reuse happen. I guess I could have called it the three W's of reuse: what, when, and who.

"Why is this an important question?" you might ask. The first answer that comes to my mind is that if you would like to build a tool to help reuse software, it would be reasonable to know: 1) what you were trying to reuse, 2) when you would be doing it, and 3) who would be using it. That is one reason, a pretty good reason, but not the only reason for asking the question "Where does reuse start?" Rhetorically, if one could understand the ramifications, implications and economic justifications of the answer to the original question, "Where does reuse start?", one would better be able to answer the question "Where should reuse start?" and "What needs to be done to make it happen?" This is the real question I think we are here to answer.

Product

If one examines the question of "Where does reuse start?" by focussing on the products being reused, one could ask "Does reuse start with code?" There is no denying that software reuse generally ends with "code". But, this still is a pretty broad statement. After all, code could be source code, object code, a high level language statement, a function, a procedure, a package, a module, or an entire program. The issue raised then is "What is the granularity of the code that you want to reuse?" The larger the granularity, the larger the "win" is in productivity. The overhead for finding, understanding and integrating a reusable software component needs to be less than design-

ing and writing the code from scratch. This supports the argument for the reuse of higher granularity objects such as software packages, modules or classes.

Just as we could debate the granularity of the object being reused, one could argue about the level of abstraction that is being manipulated. Does reuse start with a design? A design is a higher level abstraction compared to an implementation. Let me emphasize that the advantage of starting reuse from a design is that a design is at a higher level of abstraction than an implementation. Or, in other words, a design has less implementation details that constrain its applicability.

This brings out a point made in a recent paper I have been writing called "Software Reuse Rules of Thumb". In it I propose two general rules of thumb for software reuse: 1) to separate context from content and concept, and 2) to factor out commonality, or to rephrase this second rule a bit, to isolate change. If one applies the first rule of thumb, a program design, say at the detailed logic level, should have absent some (but not all) of the contextual information that will be supplied at implementation time. That is, the implementation issues, such as specific operating system or hardware dependencies, are neither part of the content, which is the algorithm or data flow nor part of the concept, which is the functional specification. I will address the second rule of thumb, factoring out commonality, later.

Before proceeding, I would like to emphasize the importance of representation, especially from a tool perspective. Remember I stated earlier that one of the reasons for looking for an answer to the question of "Where does reuse start?" was to provide a rational for building tools to assist in the reuse process. This implies that we would like a machine manipulable reusable design representation. This is not easy! But, I believe the state of the art is now evolving to a point where there are results of software reuse starting from design. The projects, that I am aware of, have been at MCC, with the DESIRE system, and at Toshiba, where in the 50 Steps per Module system, they are working on an expert system to automatically generate C, FORTRAN or Ada from low-level design data-flow charts. Furthermore, they claim success in reverse engineering existing software by synthesizing data/flow diagrams for potential reuse.

Continuing our analysis of the question "Where does reuse start?", could reuse start with a program's specification? By specification, I mean a statement of "what" a program needs to do, not "how" it is supposed to do it. There is a simple answer, yes, in limited contexts, program specifications can be reusable. But research in automatic programming tells us that this is a hard problem to extrapolate outside of narrow domains.

Speaking from personal experience, we at IBM in Owego have developed some reusable avionics specifications. When I say specifications, I mean MIL/STD/2167 System Requirements Specifications (SRS). They are highly parameterized documents full of empty tables and missing parameter values. The systems analyst, in effect, programs a new module by specifying the values in the tables of the SRS document. An application generator then reads the document and builds the data structures necessary to drive the supporting software.

Completing the waterfall model, we can ask the question on whether reuse can start with a problem definition (requirements). This is an interesting question. One might ask how? One could reason that if the same requirements can be identified as being satisfied by certain previously developed modules, then clearly those modules are candidates for reuse. Well that is a big if. It is significantly dependent on the traceability of requirements to specifications, the traceability of specifications to design, and the traceability of design into code and, also into test cases, and documentation.

Here is where a hypertext system's information web is ideal for linking these artifacts together. With a hypertext system, you can walk the beaten path to find out what code to reuse. But, there is a catch. As Ted Biggerstaff has repeatedly stated, there is no free lunch. You have to pre/engineer the artifacts to fit into the network, and spend the time and effort to create the links. Finally you need to somehow separate the context of the objects from the content. One mechanism for achieving this goal is through parameterization. Parameterization is a way to extend the domain of applicability of reusable software. Parameterization allows a single module to be generalized over a set of solutions.

To summarize, the issue we have been exploring related to the question of "Where does reuse start?" is really the question "What software artifact does reuse start with?" Part of the answer lies in the fact that we know that software reuse generally ends with the reuse of code. Where it starts depends on: 1) how much effort we want to place in developing the reusable artifact that we want to begin with, 2) how effectively we can link it to an implementation, and 3) (maybe not so obvious) how effectively we generalize the implementation.

There is a fourth dependency having to do with the process of software reuse. This is topic I will address subsequently. First I would like to reflect on the generalization issue of an implementation. One must recognize that as we progress down the waterfall model, from requirements to implementation, each artifact adds more detail. An implementation is one instantiation of a design. There could be several implementations of a design just as there could be several designs that satisfy a specification but that have different performance and resource attributes. The key is factoring out the commonality by separating the context from the concept and content. The concept becomes the *functional specification*. The content becomes a template or generic object. The context becomes possible instantiation parameters. We have identified some of the dimensions and implications related to which software artifact to start reuse with. I have concluded that code is a safe place to start and is, in most cases, the place one ends up. I also have mentioned that hypertext is the way to establish the traceability between requirements, specification, design, tests and implementation.

Process

Turning to the software development process, one could observe that most software reuse starts at the implementation phase. One could modify the software development process to include a step where, at implementation time, one would look for existing software to save having to write new code that would do the same thing. With a little luck, this usually works. But with a little foresight, this usually works better. How often is it the case that the code one wants to reuse has to be modified because either it was not implemented to exactly fit the new context it is being reused in,

or it was not implemented to provide a parameter for adapting it to a different context, or the design was such that it placed unnecessary constraints on the implementation? If the software designer had not placed the (somewhat) arbitrary design constraints, then the implementation could be used as is.

Therefore, with a little foresight, reuse might better start at design time. The implementor could then leverage off the functionality of existing implementations. This is where the bottom-up aspect of reuse meets the top-down functional decomposition aspect of most design processes. One could argue that object oriented design would eliminate this problem. Let me say that object-oriented design helps reduce the problem of the design not meeting the implementation, but parameterization still is the key for controlling this process.

One could just as easily extend the same argument for looking for reuse opportunities at design time, for the same reasons, to the specification and requirements analysis phases of the software life cycle. Again, by identifying earlier on in the software development life cycle, what is available to be reused, trade-offs can be made in the specifications, or designs can be tailored to leverage off the existing software base.

Let me now introduce somewhat of a new phase in the traditional waterfall model that has been added explicitly to support software reuse. I define domain analysis to be a generalization of requirements analysis - instead of analyzing the requirements for a specific application, the requirements of a generic application are quantified over a domain. Applying my two rules of thumb: commonality is factored out and context is separated from concept and content. Reusable objects are identified, and their context defined.

If one recognizes that the software development life cycle needs to be modified in order to inject software reuse technology, then, relating to personal experience, reuse opportunities and potential can be identified at code review time, or at design review time. If one looks at the Programming Process Architecture used in IBM, one can see these criteria called out as being integral parts of the inspection process.

But then again, instead of reuse being addressed during the software development effort, maybe reuse could start as an after thought (project follow/on). After one pass through the software development life cycle, the second time through one can begin to see the commonality between applications. Quoting Ted Biggerstaff's rules of three "If you have not built three real systems in a particular domain, you are unlikely to be able to derive the necessary details of the domain required for successful reuse in that domain." As a side point, there is a second rule of three. "Before you can reap the benefits of reuse, you need to reuse it three times." The empirical evidence I have seen to date bear this out.

A better choice for where reuse should start is at the beginning of a project (project start up). Here, the software development process can be defined, reusable software libraries can be set up and standards as well as tools developed. To share with you again my personal experience, in one large Ada project, A Computer Integrated Manufacturing (CIM) effort involving 350K SLOCS, the project had a PRL - Project Reuse Lead. He was responsible for sitting in on all design and specification reviews to identify commonality between subsystems and support the communication and application of reuse technology. Because of software reuse, factoring out commonality, the size and development effort of the project was reduced by over 20%. This is a successful example of where reuse started at the beginning of a project.

But, then again, maybe reuse could start at the end of a project (project wrap-up). I am reminded of the General Dynamics approach for developing reusable software related to an early version of the DARTS system. Here, after a project was completed, and before the design and development team was assigned to a new project, they locked everyone up in a room and wouldn't let them out until they developed an archetype of the system. That is, they recorded how and what to modify in the system so that it could be reused in the future.

While this is one approach for developing reusable software, it seems like putting the cart in front of the horse. But, then again, it is reasonable, upon the completion of any project to identify likely components to add to a reuse library.

Finally, we are all in this for the bottom line. Let me state my version of the Japanese software factory's motto: "Ask not what you can do for your software, but what your software can do for you." It makes sense, dollars and cents, to capitalize on existing software resources and expertise. But, you need to develop a business case to justify the additional cost of developing reusable software.

To summarize, the issue we have just explored related to the question of "Where does reuse start?" is really the question "Where in the software development life cycle does reuse start?" Where it starts depends on 1) how one modifies the software development process to identify opportunities for reuse, and 2) how one either modifies or extends the software life cycle to identify objects to make reusable. The bottom/line is that software reuse is a good example of software engineering discipline.

Personnel

Turning to the last dimension I identified related to the question of "Where does Reuse Start?", we will focus on the key players in the reuse ball game. The first player to come to bat is the programmer. Does reuse start with a programmer? Most programmers are responsible for the design and implementation of software. If they can identify a shortcut to make their job easier, or to make them appear more productive to their management, then they probably will be motivated to reuse software. But, while programmers might be inclined to reuse software if it was fun, or it was the path of least resistance, or if they are told to, the real issue is "Who is going to create the software to reuse in the first place?" There needs to be a critical mass of quality software for programmers to draw upon in order for them to fully subscribe to the reuse paradigm! So, how do we bootstrap the system?

Maybe managers can instill a more altruistic attitude on their programmers. This, of course, becomes a question of budget cost and schedule risks associated with the extra time and effort needed to make things reusable.

Reuse is a long term investment. Maybe the expense of developing reusable software

should be spread across a project! With reuse raise to the project level, there would higher potential for a larger return on investment, plus more insight and experience in prioritizing what should be made reusable. Again, there is no free lunch, A project manager would have to authorize the cost. But project management is generally rewarded for getting a job done on time and under budget. There is no motivation for making the next project look good. This shortsightedness needs to be resolved with top management.

Indeed, this is the case, both here and abroad. At NTT, GTE, IBM, TRW, to name a few companies, reuse incorporation and deposition objectives are being set. For instance at NTT, top management has set a reuse ratio goal of 20% on all new projects, with a deposition ratio quota of 5%. That is, all new programs ideally should consist of at least 20% source code from the reuse library and all new programs should try and deposit at least 5% of their source code to the reuse library (subject to the acceptance guidelines, constraints, and ultimate approval of the Reuse Committee).

But, upper management edicting reuse to happen doesn't insure success. That is why there is a strong argument for reuse to start in the classroom (educator). The education system, while it is good at teaching theory, might embrace a little more of the engineering discipline and teach software building block construction or composition of programs. Courses are needed in domain analysis, application generator construction, and parameterized programming, as well as the availability of pre-fabricated, off/the shelf components structured to facilitate the construction of new applications in a classroom setting. Again, critical mass is needed to bootstrap the system.

Besides the reuse mind set, maybe reuse should start with a tool set (tool developer). Personally, I do not see the need for exotic and elaborate tools to support reuse. Although, I am biased towards using a multi-media hypertext system for the capture and representation of domain knowledge, which I consider crucial to understanding what and how to reuse software.

Have I run out of people who possibly could start the reuse ball rolling? Have I saved my heavy hitters for last? Should reuse start with the customer? It depends on the customer! A large

customer, like the Department of Defense, could easily demand certain reuse requirements be met. Of course, there might be a small initial overhead cost associated with getting the ball rolling, but once the system was primed, once application domains were populated with certified, parameterized, well documented, reusable components, then long term benefits could be reaped.

I have added the salesperson to this list of individuals who could play a role in determining where reuse might start. The reason is that if a salesperson knows the marketplace and knows potential customers, then they could play a key role in building the business case necessary to justify the capitalization of software for reuse.

Finally, I have added the systems analyst as being a person who possibly could be instrumental in starting software reuse. I admit, he joined the team late, but he turns out to be a clutch player. Back to the issue of putting the horse in front of the cart. Before you can reuse software, you need software to reuse. Who are you going to call? The domain analysts! Who are the most qualified individuals in an organization to 1) analyze a problem domain, 2) determine logical subsystems and functions, and 3) determine the contents or requirements of modules and anticipate the different contexts that they might be applied under? The systems analysts. They have made life so difficult for some of us programmers in the past by providing incomplete or inconsistent or, worse yet, too detailed specifications. This is a wonderful opportunity to work together toward a common goal.

To summarize, the issue we have been exploring related to the question of "Where does reuse start?" has been identifying the roles played by certain individuals in an organization related to making software reuse happen. In retrospect, several of the key players had non-technical roles in the game! A point that bears distinction and should come as no surprise.

Summary

In conclusion, the goal of my presentation was to bring to light issues surrounding software reuse. To force you to question what you might have accepted on blind faith. I have probably raised more questions than I have answered, but, that is good. Hopefully it will provide you oppor-

tunities for discussion. Finally, I have shown, as a wise old owl once stated, "It is not what you know, but who, you know?" that often is necessary for success. Software reuse is no exception to this rule. Software reuse is a people issue as well as a technology issue

APPENDIX C

Workshop Attendees

Gregory Aharonian
Source Translation & Optimization
P.O. Box 404
Belmont, MA 02178
(617) 489-3727

Dennis M. Ahern
Senior Engineer
Westinghouse Electric Corporation
Aerospace Software Engineering
P.O. Box 746
MS 432
Baltimore, MD 21203
(301) 993-6234
DAHERN@SIMTEL20.ARMY.MIL

Rich Armour
United States Air Force
HQ USAF/SCW
Washington, DC 20330-5790
(202) 694-8890

Brian Baker
Department of the Navy
Chief of Naval Operations (OP-945D6)
Washington, DC 20305

James Baldo, Jr.
Research Staff Member
Institute for Defense Analyses
Computer and Software ENgineering Division
1801 N. Beauregard St.
Alexandria, VA 22311-1772
(703) 824-5516
baldo@ida.org

Bruce H. Barnes
National Science Foundation
Information, Robotics, and Intelligent Systems
1800 G Street N.W.
Washington, DC 20550
bbarnes@note.nsf.gov

Edward W. Beaver
Westinghouse Electric Corporation
Defense & Electronics Division
P.O. Box 746, MS 432
Baltimore, MD 21203
(301) 765-3926

Ted Biggerstaff
Microelectronics & Computer Technology Corp.
9390 Research Blvd.
Kaleido II Bldg.
Austin, TX 78759
(512) 339-3600
big@mcc.com

Christine Braun
Contel Technology Center
12015 Lee Jackson Highway
Fairfax, VA 22033-3346
(703) 818-4475
braun@ctc.contel.com

Patrick Carroll
Resident Affiliate
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Joel Cohen
GTE Government Systems Corp.
Strategic Electronics Defense Division
National Center Systems Directorate
1700 Research Boulevard
Rockville, MD 20850
(301) 294-8400
cohen_jm%ncsd.decnet@getwd.arpa

Sholom Cohen
Member of the Technical Staff
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-5872
sgc@sei.cmu.edu

Liesbeth Dusink
Delft University of Technology
Faculty of Mathematics and Informatics
Julianalaan 132
2628 BL Delft
NETHERLANDS
betje@dutinf

Steve Edwards
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311-1772
(703) 845-3536
edwards@ida.org

Richard E. Fairley
Professor
George Mason University
School of Info Technology
SITE, Room 203
4400 University Drive
Fairfax, VA 22030
(703) 764-6195
fairley@gmu.vax.bitnet

Fred J. Foster
Staff Assistant
United States Air Force
Office of the Secretary of Defense
Director, Operational Test and Evaluation
The Pentagon
Washington, DC 20301-1700
(202) 694-2153

Ernesto Guerrieri
Softech, Inc.
460 Totten Pond Road
Waltham, MA 02154-1960
(617) 890-6900
ernesto@ajpo.sei.cmu.edu

Harley Ham
Naval Avionics Center
NAC-825
6000 E. 21st Street
Indianapolis, IN 46219-2189
(317) 351-4457

James A. Hess
Resident Affiliate
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-5851
jah@sei.cmu.edu

Daniel E. Hocking
Computer Scientist
AIRMICS
Computer and Information Science Div.
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800
(404) 894-3110
hocking@airmics.army.mil

Rick Holbert
United States Air Force
HQ AFSC/PLR
Andrews Air Force Base
Washington, DC 20334-5000

Robert Holibaugh
Project Leader
Software Engineering Institute
Methods Program
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-6750
rrh@sei.cmu.edu

Harry F. Joiner
Telos Federal Systems
55 N. Gilbert Street
Shrewsbury, NJ 07702
(201) 530-8444

Kyo Kang
Member of the Technical Staff
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-6415
kck@sei.cmu.edu

Beverly Kitaoka
Science Applications International Corporation
311 Park Place Blvd.
Clearwater, FL 34619
(813) 799-0663

Larry Latour
Assistant Professor
University of Maine
Department of Computer Science
Neville Hall
Orono, ME 04469-0122
(207) 581-3941

Ken Lee
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-7702
kl@sei.cmu.edu

Stanley H. Levine
710 Carol Avenue
Ocean, NJ 07712
(201) 544-3098

Jim Lund
United States Air Force
AFATL/FXG
Eglin AFB, FL 32542-5434
lund@uv4.eglin.af.mil

Rod Moyes
United States Air Force
OOALC
MMETI
Hill AFB, UT 84056
(801) 777-7703

William Novak
Resident Affiliate
Software Engineering Institute
Software Methods
5000 Forbes Avenue
Pittsburgh, PA 15213
wen@sei.cmu.edu

Phil Palatt
Senior Staff Engineer
Dynamics Research Corp.
Systems Division
1755 Jeff. Davis Hwy. #802
Arlington, VA 22202
(703) 521-3812

Constance Palmer
Senior Engineer
McDonnell Douglas
Missile Systems Company
Dept. E434, Mail Code 0922232
P.O. Box 516
St. Louis, MO 63166
(314) 925-7930

Jim Perry
Resident Affiliate
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-7744
perry@sei.cmu.edu

A. Spencer Peterson
Member of the Technical Staff
Software Engineering Institute
Software Methods
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-7608
asp@sei.cmu.edu

Charles Plinta
Member of the Technical Staff
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-7771
cpp@sei.cmu.edu

Ruth Rudolph
Computer Sciences Corporation
Defense Systems
304 West Route 38, Box N
Moorestown, NJ 08057
(609) 234-1100 x2237

Theodore Ruegsegger
Softech, Inc.
460 Totten Pond Road
Waltham, MA 02154-1960

Ruth J. Shapiro
Resident Affiliate
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 236-6398
rjs@sei.cmu.edu

Mary Shaw
Professor of Computer Science
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-7731
shaw@sei.cmu.edu

James Solderitsch
Research Scientist
UNISYS Corporation
Defense Systems
P.O. Box 517
Paoli, PA 19301-0517
(215) 648-7376
jjs@prc.unisys.com

William L. Sweet
Manager, Industry Sector Operations
Software Engineering Institute
Technology Transition
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-7706
ws@sei.cmu.edu

Will Tracz
OBM Systems Integration Division
Mail Drop 0210
Route 17C
Owego, NY 13827
(607) 751-6731

Roger Van Scoy
Member of the Technical Staff
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-7620
rlvs@sei.cmu.edu

Terry Vogelsong
Department of the Army
Information Systems Software Development Center
Attn: ASQBI-WRC STOP H-4 (Terry Vogelsong)
Fort Belvoir, VA 22060-5456
(703) 756-5202

Paul A. Wilbur
Principal Engineer
Teledyne Brown Engineering
Cummins Research Park
300 Sparkman Dr. NW
Huntsville, AL 37807-7007
(205) 726-1505

APPENDIX D

Workshop Position Papers

SOCIAL AND ECONOMIC PROBLEMS WITH DEFENSE SOFTWARE REUSE

Gregory Aharonian
Source Translation & Optimization
P.O. Box 404, Belmont, MA 02178
617-489-3727

Software reuse is not a technical problem. The current approach to fostering software reuse, such as in the STARS project and at many companies, is to concentrate on developing new software tools and methodologies. However, the greatest impediments to software reuse are not technical, so that any new tools and methodologies will have marginal impact.

Rather the problems that should be addressed deal with economics, management, technology transfer, training, legal issues, politics, tradition, and the continual advancements in technology. Until these issues are dealt with, it is unlikely that companies and the DoD will ever achieve cost reductions through software reuse.

What follows are questions concerning the non-technical aspects that have to be answered to achieve a successful software reuse effort, be it for a company, or the whole DoD. Unfortunately, at most sites around the country, the answer to most of these questions is either "No", or "I don't know".

IGNORING EXISTING RESOURCES

Have the current software resources at the company been cataloged in printed or machine readable form? Where in the company is developed software archived?

Have external sources of software resources available for reuse at the company been cataloged? Where in the country are available software resources kept? How aware are company programmers of external software resources?

How can these master directories be prepared? What librarian skills are needed (for classification and abstracting)? Is there a charge number to put someone on this project full time? What help can the company librarian offer?

How well are existing software resources being reused? How can current reuse rates be measured, and is the information being collected?

What information should be collected for each program? Can this information be mapped into a database schema? Should the database schema be used with a traditional database system or an expert system, with or without natural language query capabilities (does the DoD have a natural language front-end that can be reused for developing a retrieval system)?

Why hasn't the DoD (using DTIC, DACS, DARIS, ISEC, DARPA, SEI, DCA, or ESD) funded the collation of a directory of the 10,000+ programs available for reuse throughout the defense community? Does this mean that the DoD doesn't want people reusing its' software?

Can software metrics be used to compare software packages providing the same capability? How can the best mathematical library in Ada be selected from two or three repository offerings? Does the library with the highest quality code from a software engineering viewpoint also have the most accurate code numerically? Does the company's reuse staff have the expertise to answer these questions?

ECONOMICS

Do the company's programmers have incentives to reuse software (are they paid for lines-of-code, or productivity)? Are managers trained to assess a programmer's software reuse activities?

Do the programmers have tools to quickly perform cost/benefit analysis at various stages of the life-cycle to know when to reuse software?

Has the company collected financial data on previous software projects to use in cost/benefit analysis tools? What are the financial factors that contribute to the cost of reusing software?

How can technology be factored into or out of cost/benefits models? For example, 1986 Ada compilers running on 10 Mhz cacheless 80286 systems run about ten times slower than 1989 Ada compilers running on 30 Mhz cached 80386 machines.

Do the company's managers know how to create incentives for programmers to reuse software? Are managers rewarded if they save money through reuse?

Should consultants be used to help the company's programmers acquire and reuse software? How much should be spent on consultants, and how can their activities be measured to see if the company is saving money?

Do the company's managers know how to bid proposals that are based on some estimated level of software reuse?

Is the company being rewarded by the DoD for reducing costs by reusing software? Do contracts have incentive clauses, and can DoD program managers earn credit for managing programs that spend less money?

Will the company be penalized by the DoD for not reusing software? If the DoD wanted the company to reuse software, why don't they require companies to report on applicable reusable software in submitted proposals?

How can the DoD coordinate software reuse nationally? What does it mean to have a national software reuse policy? Does the foundation of this policy - the mandatory reuse of publicly funded, publicly owned software available to everyone (even competitors) from repositories - sound marketed oriented or command oriented? What kind of economics does software reuse impose on a company, or a country?

In an era of corporate restructuring and leverage buyouts, which has and will be happening in the defense community, how can the long term funding needs of software reuse be made compatible with the short term profit seeking to satisfy creditors and Wall Street?

How can the company successfully transfer reusable software technology from their defense activities to non-defense activities? How can DoD software activities be channeled to fight both a military war with the Warsaw Pact, and a economic war with Europe and Japan?

MANAGEMENT

How serious is the DoD about software reuse? They've booted STARS back and forth, cut funding to STARS, supported parallel (and duplicative) software reuse efforts, and haven't put a general or admiral in charge. Will the company get in trouble for aggressively promoting software reuse?

Are the company's managers given the financial and administrative resources to establish software reuse programs? Is upper management willing to invest for many years before realizing any gains?

Is upper management or the DoD willing to allocating funds to have people full-time maintaining the reuse effort at the company? If such activities cannot be charged to existing contracts, will the company allow the activities to be funded out of administrative funds. Money is provided for IRAD in software - how about IRAD money for transferring IRAD software results?

Do any of the company's managers have experience to manage a software reuse program? Do they have the requisite training in software development, technology transfer, cost/benefit analysis, systems analysis, applications programming, and library science?

Is the company advertising for positions such as managers of software reuse? Is the need for such people understood? What qualifications should this type of manager have? Where in the corporate hierarchy does the manager and his staff report to? Can a software reuse manager force software to be used on specific projects? How absolute and how great is his authority? What kind of supporting staff is needed?

Where is the line drawn on a software reuse staff's activities? Should they be a service, acquiring and handing off software on demand, or should they use their expertise to also integrate the software and solve the problem?

Should there be a distinct software reuse group, or should their activities be distributed across all programming groups at the company?

Have the company's programmers been surveyed to determine what kinds of software they need and are interested in reusing? Have they also been surveyed to acquire their suggestions on how the company can best reuse software?

Do programmer's have permission to visit, on company time, local government and unviarsity facilities to discover sources of reusable software? Will programmers be given inhouse training to transfer software technology from domains they have minimal experience? Reusing software implies reusing the technology being implemented in the software. For example, defense engineers have wasted much time and money reinventing capabilities with neural networks that were already available in existing statistics and physics algorithms, with which they probably have had little formal contact or training. How can the company avoid this waste, and what kind of training will be needed?

AVOIDING THE N-I-H SYNDROME

Is software being used from all sources, or is the company or command being parochial? Can and is other companies' public domain software be reused? Is the Navy using any of the Air Force's software, and vice-versa? Can NASA software be reused in DoD projects, and why are both agencies funding software reuse environments to be use by the same contractors?

Is the testing-and-verification excuse being used to avoid reusing external sources of software? On the other hand, is testing and verification of reusable software too costly for those interested in reusing software?

How about all of the disparaging gossip about repository software? Will this discourage upper management support for reusing repository software? For example, supposedly the official STARS editor is so large when compiled that it won't fit into 640K of PC memory, requiring extra memory on expansion boards. Is this reflective of the rest of the repository software?

In an era of competitive software marketing, if a program is good enough to be of interest to many people, it is good enough to be marketed commercially for a profit. Does this mean that the only kind of software being donated to public repositories is software of limited use to others, and therefore not worth reusing? Will competitors try to sabotage the company by offering reusable software that is more trouble to reuse than it is worth in savings?

Will the Ada zealots bring down plagues on the company if we minimally translate good algorithms from Fortran and C?

LEGAL

What is the cost burden of dealing with software in public repositories that the original developing company has partial rights to?

Does reusing software in certain configurations violate known and unknown patents? Will companies be liable for actions of uninformed programmers? Will companies need lawyers as part of their software reuse staff?

Who is liable for faults attributed to acquired repository software components? At which point in modifying external software, does the burden of reliability pass from the original developer to the modifying company?

How well should software be tested before being contributed to repositories? How well can software be tested in general? Will other companies provide access to the training data used to develop and test the algorithms that they have placed in the repositories?

How willing is the DoD management to support publicly available software repositories, knowing that all of the software will quickly make its way to Moscow?

Version 2 (V2)

Brian Baker/OP-945D6
Anna Deeds/PMS-412
X56311/X28204
29 Mar 89

29/30 Mar 89 Ad Hoc DoD Software Reuse Strategy Working Group
Institute for Defense Analyses, Falls Church, Va

Position paper: Industrial Policy and Software Reuse: A Systems
Approach

Reuse is one of several modern, revolutionary, interrelated and interdependent software productivity techniques. Productivity, not specifically reuse, should be our overarching concern.

Reuse offers perhaps the best opportunity for evolving software programming into a productive, hard engineering discipline based on standard engineering designs, algorithms, and reusable code.

Software programmers should be held to the same standards of timeliness and productivity that we hold other engineers to.

Reuse should be looked at systematically within the matrix of evolving library technology, other interdependent software engineering technologies, corporate micro economics, and governmental macro economics.

Most software problems are management, not technical, problems.

Therefore, software productivity, and the subset of reuse strategy, needs to be addressed as a management problem in the large scale.

What is needed is an industrial policy or national strategy for software productivity to be implemented through procurement policy.

Advanced software engineering processes and technology will continue to be one of America's most important technological-strategic assets.

The distinction between military mission-critical technologies and civilian technologies is increasingly blurring.

Key questions: On what scale do we implement reuse, particularly for complex R&D-based mission-critical systems? Should reuse be implemented on the corporate scale, or on a governmental scale? Or both?

Since reuse can only be achieved through a merging of library technology and software engineering processes, the proper scope for reuse is probably at the corporate, rather than governmental, level.

Why should the government create massive software libraries if the technology for reuse is so interdependent on internal, corporate development processes?

It only makes sense for the government to establish libraries if it does its own software development. It cannot and does not do this for large-scale mission-critical systems. Perhaps governmental libraries make sense for developing small-scale, mission-support systems.

It is imaginable, and therefore possible, that software development will someday be done using artificial intelligence with integrated libraries and increasingly automated software engineering processes. If so, stand-alone, governmental libraries (or more likely, repositories) will not be sufficient for this discipline.

Governmental libraries will probably always tend toward obsolescence. Government does not have as much incentive as industry for maintaining state-of-the-art libraries.

Particularly for large-scale mission-critical systems, liability problems associated with GFI, verifiability and cataloging issues, and evolving software engineering technology make governmental libraries a monumentally complex and expensive proposition. On the other hand, corporations have a perfect incentive to do this.

Government should not tell corporations how to reuse software or make them use governmental libraries. If reuse makes sense, they will do it. They do not need us to make them do this. Double-billing for reused software should be ameliorated by increased competition and increased emphasis on establishing productivity baselines and measures.

As corporations continue to increase their technological and economic resources, and merge, their political, economic and technological power will probably eventually transcend that of many nation states and make national boundaries irrelevant.

If the Chinese and Russians become liberal, pluralist, capitalistic/democratic, the global free-market, capitalist economy will inevitably lead to worldwide competition on an as-yet unimagined scale.

Future long-lived multinational corporations will probably tend to have extremely secure, fortified, state-of-the-art, multiple

domain-specific libraries since these libraries will probably be indistinguishable from their software engineering centers.

Technical data libraries will be afforded the same security as software engineering centers, probably more.

Corporate software engineering centers, including distributed mainframe-based multi-configuration workstations, will probably be integrated with domain-specific technical data libraries which would be used by the software engineer in designing applications and reusing relevant components.

Currently, software is more responsive than hardware (materials) to changes in the threat, e.g., embedded mission-profile computers.

Since software (and associated technical data) is a significant corporate (and national) asset, corporations will probably place a very high price on making it available to governmental libraries. This approach to reuse becomes expensive.

Since each corporation will have a vested interest in any library interoperability issues (to assist teaming arrangements), including graphical user interfaces, hypertext formats, i.e., the "standards" issue, we should leave industry to initiate any work in this area.

Industry will probably not be responsive to governmental "standards" groups in the library/reuse area but we should be prepared to assist any work they wish to pursue.

As a preliminary suggestion, reuse libraries should employ the card catalog principle implemented in public libraries, i.e., use standard information block with (1) author; (2) title; and (3) subject.

Cost-plus contracts also provide a disincentive for reuse.

Fixed-Price contracts with incentive fees (FPIF) for advanced or mission-critical applications will only work within revised acquisition and life-cycle maintenance policies.

FPIF contracts will only work within revised profit regulations. Corporations should be allowed to make any percentage profit on government contracts and be incentivized to contain costs.

The marketplace and increasingly sophisticated competition should regulate profits, not artificial, national government policies which may increasingly conflict with relatively unbounded international economic norms.

FPIF contracts are needed to force industry to treat software programming as a hard engineering discipline. Unless we try, it will not happen by itself.

Therefore, the government needs software engineers to draft RFPs and evaluate proposals based on certain software engineering disciplines evidenced in proposals.

Incentive awards should be made for software readability, clear documentation, object-oriented programming, user-friendly man-machine interfaces and productivity.

In conjunction with this, corporations should retain all rights to their software, just as they would any other piece of proprietary hardware.

Procurement of advanced, mission-critical systems should begin with rapid prototyping and modeling by two or more competitors for proof of concept. Focus should be on establishing stringent productivity measures and baselines for incentive awards.

Any additional time that the rapid prototyping/proof of concept phase would add to the acquisition process should be balanced by shortened Full Scale Engineering Development (FSED).

During the life of a contract, productivity standards should be increased from a baseline such that the contractor could not but help to employ reuse or other productivity strategies.

An object-oriented programming methodology should be required for MIL-STD 2167A. Associated with this, the government needs to train personnel in software engineering and object-oriented programming.

Contract specifications should concentrate on performance and functionality. Government acquisition managers must remain flexible and open to new corporate software engineering practices if they make sense.

After rapid prototyping, competitive FPIF leader-follower FSED contracts should be issued.

Competitive second sourcing should be considered for long-term full-production contracts. This applies to system hardware and software.

Our present method of separating software development (contractor-based) and life-cycle maintenance (government-based) activities will probably not work for increasingly complex systems, unless the most-likely life-cycle maintenance activity is involved during the developmental phase.

By separating these two software phases (often between industry and government), government is in effect paying for the same software twice: once in development, and again when it trains and staffs in-house personnel to try to understand, and then modify, the same software.

Government should compete long-term life-cycle maintenance just as it does research and development.

Industrially-funded, highly-professional, governmental, domain-specific software support and engineering centers should be established in the field to compete with original contractors for life-cycle maintenance of software. But the original developer's library or software engineering center should be readily accessible to governmental personnel. Good examples of governmental software support centers are the Fleet Combat Direction Systems Support Centers and the Marine Corps Tactical Software Support Activity.

In conjunction with this this, the government needs to train more personnel in the software engineering discipline, e.g., send them to the Software Engineering Institute (SEI).

Governmental research and development centers would then have a choice between contractor-supplied and governmental (in-house) maintenance. Major enhancements or modifications to software (driven by operational requirements) would continue to be funded by the R&D centers.

Making Software Reuse Cost Effective

Bruce H. Barnes
 Division of Information, Robotics, and Intelligent Systems
 National Science Foundation
 1800 G Street NW
 Washington, D.C. 20550
 (Internet: bbarnes@note.nsf.gov)

Terry B. Bollinger
 Contel Technology Center
 12015 Lee Jackson Highway
 Fairfax, Virginia 22033-3346
 (Internet: terry@ctc.contel.com)

To be cost effective, software reuse must be recognized as having the same cost and risk characteristics as financial investment. This paper gives an overview of a set of analytical methods that are based on the precept that reuse is a form of investment, and it describes how such methods could provide a more uniform, rational basis for customizing the application of reuse technologies to the specific situations of developers, projects, and organizations.

sists of any outlay of current resources or labor hours in hopes of future reductions in costs. Additionally, reuse investments consist only those outlays which do not directly contribute to the development objectives of the project. They are thus a form of overhead, at least as far as the originating project is concerned.

A project or activity which makes reuse investments is referred to in this paper as a *producer* project, since it functionally has the role of supplying a reusable "product" to one or more later projects or activities. It is important to note that, by definition, a producer activity does not accrue any direct cost benefits from building reusable products. *Consumer* projects are ones which reduce their total development costs by replacing one or more of their development activities with the acquisition of a reusable product that originated in a producer project.

1. Introduction

The central premise of this paper is that to be cost effective, software reuse must be recognized as having the same cost and risk characteristics as financial investment. The rationale for this assertion is shown graphically in Figure 1. A *reuse investment* con-

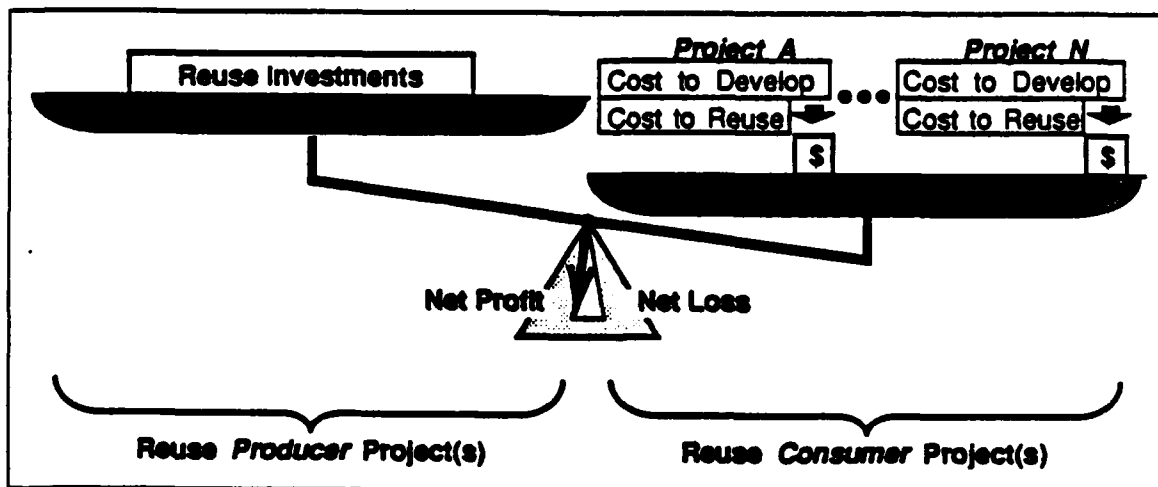


Figure 1 - Software Reuse as an Investment

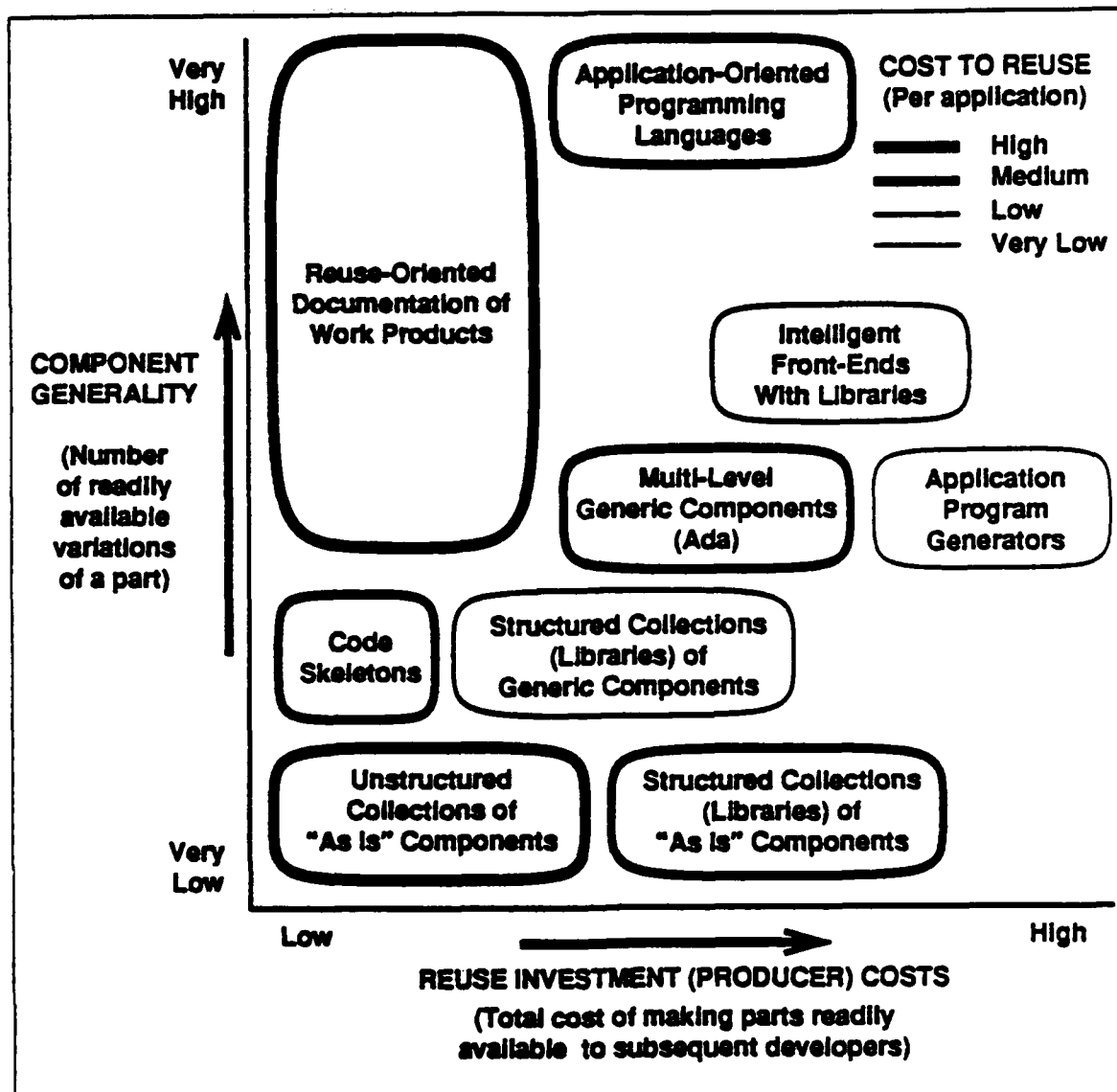


Figure 2 - Cost-Based Selection of Reuse Technologies

Just as is the case in sales of commercial software products, a reuse producer can see a "profit" only when there are enough customers to make overhead costs of reuse investment worthwhile. Additionally, it is very important that consumer activities achieve significant savings in development costs, else the number of instances of reuse will need to be so large that it is unlikely that a net profit can be achieved.

The producer/consumer model of reuse is rich in both managerial and technical implications. One obvious managerial implication is that organizations which fail to provide some form of payback incentives to produc-

er projects are unlikely to succeed in making reuse an integral part of their development process, since producers will be penalized for overhead expenditures which do not directly benefit their development needs.

From a technical viewpoint, Figure 2 shows how a producer/consumer model of reuse can be used to support an integrated view of a very diverse set of reuse technologies. The key idea of Figure 2 is that selection of a specific reuse technology should not simply be a matter of personal preference, but should instead be based on an analysis of the specific needs and constraints of projects. Cost-based selection of reuse

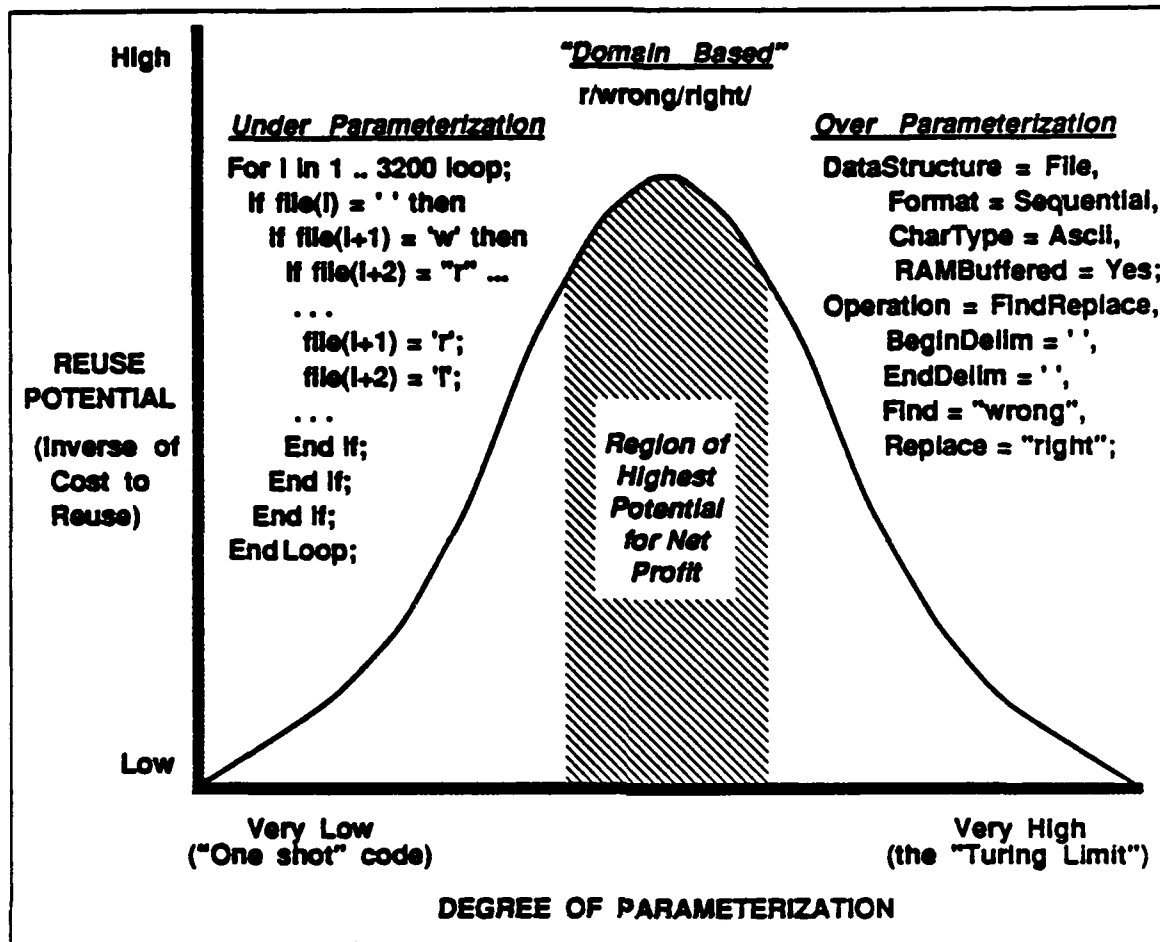


Figure 3 – The Limits of Parameterization in Reuse

technologies need not be restricted to only the large-scale granularity of projects; it can also be applied in a fine-grained approach as a way of integrating multiple reuse technologies into a single project. In fine-grained applications, cost-based classification of reuse methods tends to lead to the selection of low levels of reuse investment for components with only a moderate likelihood of being reused, and high levels of investment whenever a strong "market" of consumer activity needs can be clearly identified.

Fine-grained application of cost methods to reuse requires more specific methods for analyzing and specifying the potential reuse payoffs of components. Figure 3 gives an example of such a fine-grained approach for the well-known (and deceptively simple) concept of module parameterization, in which software modules are made more reusable by increasing the number and types of pa-

rameters that can be used to control their behavior. Intuitively, one might tend to assume that if performance issues could be ignored, increasing the level of parameterization would nearly always lead to increases in the reusability of a module. For reasons which can only be briefly mentioned here, this turns out not to be the case. Very high levels of parameterization tend to approach what the authors refer to as the "Turing limit," which is the point at which the parameterization become so extensive in scope that the effective equivalent of a general-purpose Turing machine is created. At best, a parameterization scheme that has reached the Turing limit will be at least as complex as a general-purpose programming language, and could easily be far more complex; its reuse potential is thus effectively nil, since redeveloping the part in the original programming language would be less costly than at-

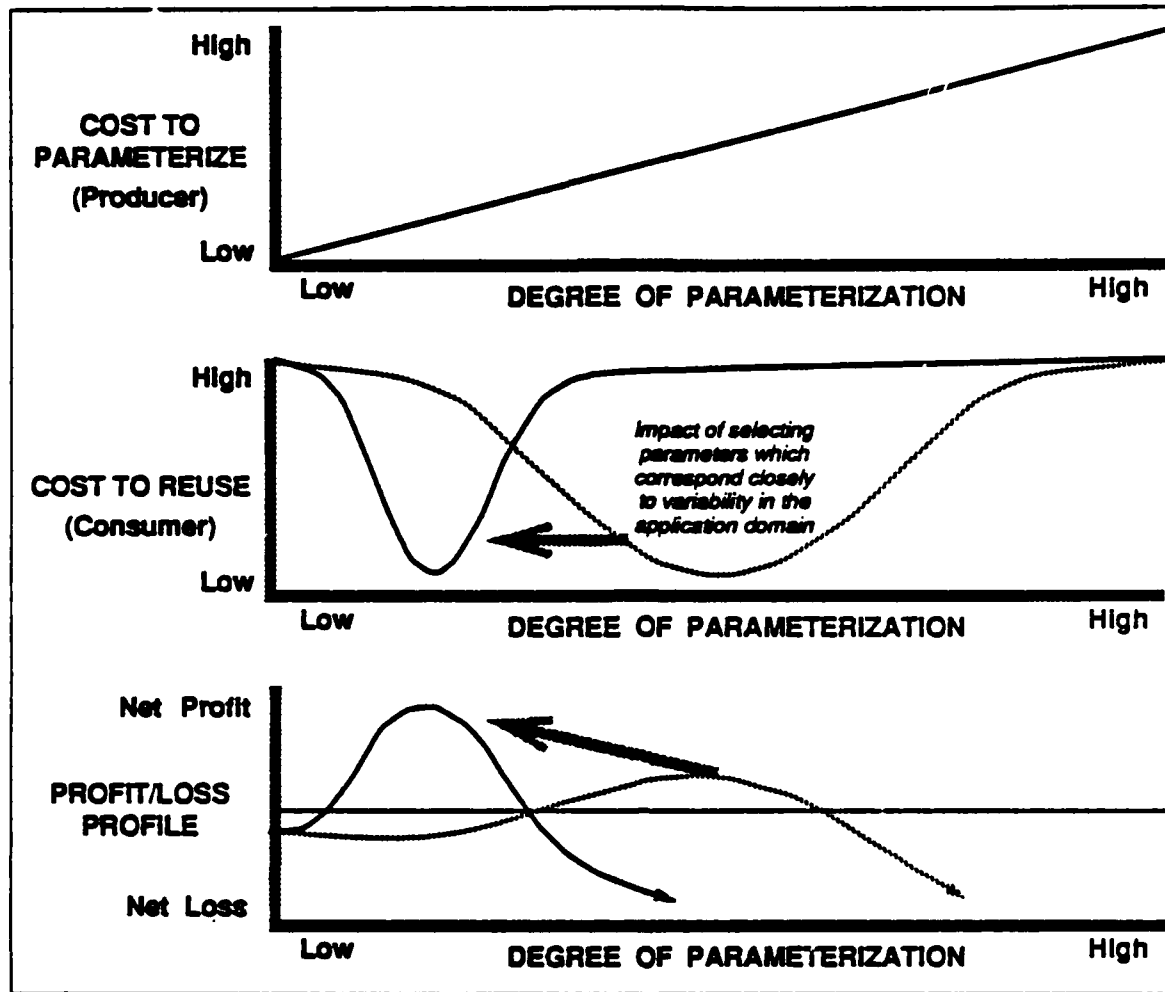


Figure 4 — Profit/Loss Characteristics of Parameterization

tempting to determine the set of input parameters needed to customize the "reusable" version of the component.

Instead, the best payoff in parameterization comes through finding combinations of parameters which in some way "cover" the most likely forms of variability of the application domain. Just as many manufacturing disciplines have developed sets of complementary parts which can be adjusted and combined to produce a very wide range of useful products, a good software parameterization scheme is one in which the features that can be changes are the very ones which are most likely to change whenever the component is reused in a consumer activity.

This idea of optimizing the selection of parameters based on the best available knowl-

edge of how the problem domain varies is shown graphically in Figure 4. Because the cost of adding new parameters to a component is roughly linear with the increase in the number of parameters, it is obviously beneficial to "skew" the selection of parameters in favor of those features which a formal or informal analysis of the problem domain indicate are most likely to benefit consumer activities. What may not be as obvious is how much of a cost impact such selections can make; as shown in the figure, a good initial selection of parameterizations can result in a substantial increase in the net profitability of that reuse investment. A little bit of explicit analysis and consideration of the problem domain before beginning the parameterization process can thus be a highly beneficial activity, one which can make it far more likely that a large number of con-

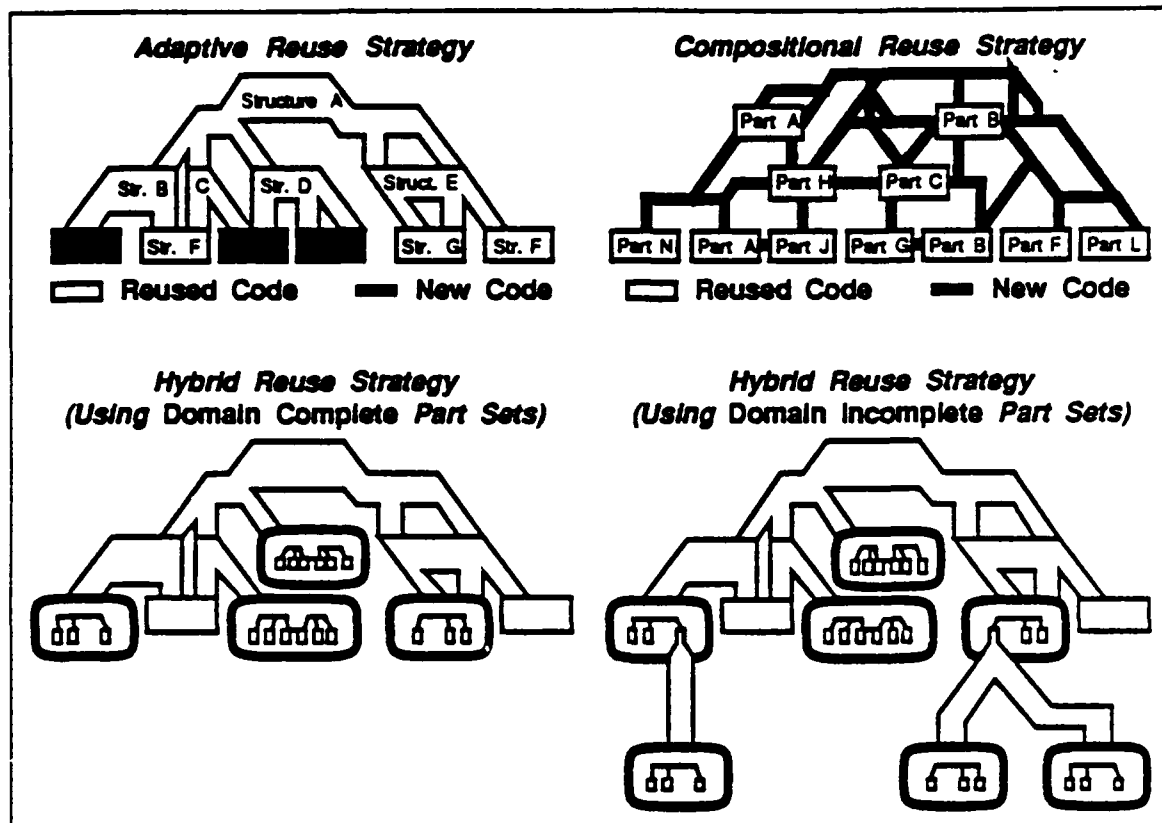


Figure 5 – Major Strategies for Applying Reuse Technologies

sumer activities will be able to make effective use of the part.

To perform the kinds of optimization just described for reuse parameterization, a producer project must be able to "divide and conquer" in its analysis of how to maximize the reuse potential of a system. Figure 5 shows one approach to the reuse version of the divide-and-conquer problem. The Figure 5 approach is based on the idea that all consumer activities must deal with two types of code: "old" code that is being reused to reduce costs, and "new" code whose purpose is to *customize* the behavior of the old parts to meet a new application. Reuse schemes may then be classified in terms of how they "mix" old and new products to create new systems.

There are two fundamental approaches to this problem. In the *adaptive* approach, the support structure is kept as stable as possible, while new code is added at the lower levels of the structure. In the *compositional* approach, new code is used to "glue togeth-

er" (compose) old code that is in the form of discrete, functionally simple modules. These two fundamental reuse schemes can be combined to form *hybrid* approaches which provide powerful analytical frameworks for understanding and optimizing the reuse potential of a producer system.

This paper is only a brief summary of a number of techniques and issues which the authors would like to suggest as being important in making reuse a widespread, integral part of the software engineering process. Although some of the issues are quite complex, there is good reason to believe that a cost-based approach to understanding, organizing, and selecting reuse technologies can provide important benefits to the software industry, in both the short and long term.

POSITION STATEMENT

**Software Reuse in Practice Workshop
Software Engineering Institute
July 11-13, 1989**

by

**Richard E. Fairley
Professor of Information Technology
George Mason University
Fairfax, VA 22030
(703) 764-6195**

Area of Emphasis: Organizational/Economic Issues

We are currently investigating incentives for reuse of Ada components for the AIRMICS organization of the U.S. Army through a subcontract from Martin Marietta, Huntsville to George Mason University. The viewpoint we are pursuing can be summarized as follows:

"Suppose the technical and legal problems of reuse were solved. What disincentives would have to be removed and what incentives introduced to make reuse of Ada components a widespread and cost effective practice?"

The main emphasis of our work is on issues of development methodology, organizational structure, organizational behavior, and economic modeling of reuse.

Although the main thrust of this work is oriented to incentives for reuse of Ada components, issues related to methodology, organization, and economics are similar for all types of software work products (requirements, design specs, code, test plans) and all types of implementation languages. We thus expect our results to be applicable in a wide variety of situations.

In the area of methodology, we are exploring the implications for reuse of object-oriented development and the "family of systems" approach of David Parnas and colleagues. There are obviously strong interactions among domain analysis, development methodology, and implementation issues. We are attempting to take those interactions into account. The strong interactions among methodology, organizational concerns, and economic modeling of reuse may be less apparent, but are equally important and in fact provide the motivation for our interest in development methodology.

We are focusing a great deal of attention on organizational and economic issues of reuse. In the long run, these issues may be more difficult and more critical to successful reuse programs than the issues of technology or methodology. We are examining issues of organizational structure to facilitate both reuse in the small and reuse in the large. We are also distinguishing between ad hoc reuse and formalized reuse programs.

Issues of organizational behavior are concerned with motivation and individualized incentives for reuse. We are investigating these issues at various organizational levels, to include programmers, team leaders, project managers, department managers, executive officers, and customers/clients. Motivational incentives have been investigated by others within the context of technology transfer, and many of the results from those investigations are directly applicable to software reuse.

Economic considerations are the ultimate test of reuse. Efforts in technology, methodology, organization, and legal concerns will be of little consequence unless it can be demonstrated that reuse increases programmer productivity and/or the quality of software. We are conducting a survey of existing cost models and software metrics programs to determine the current state of cost modeling for reuse.

We are also developing cost models that account for factors such as the increased cost of developing a reusable software component, the anticipated number of reuse instances for the component, the amount of modification effort required to reuse the component, and the percent of the component that will be reused in both modified and unmodified forms.

In addition, we are incorporating factors that account for the increased reliability of a reused component over a newly developed component. Increased reliability has several distinct effects: saving of the increased cost for developing a new component of equivalent proven reliability, the reduced maintenance effort for a proven component, and the psychological effect of using a component of demonstrated reliability.

Metrics for reuse is another aspect of economic modeling that we are examining. We are developing a set of recommended metrics to track the time and effort required to develop reusable components, incorporate reusable components into new systems, and support the infrastructure of reuse libraries, reuse personnel, and reuse histories.

The final aspect of our work involves case studies in reuse. We have identified several reuse projects that span a variety of approaches to reuse. We will examine these projects to determine factors that contributed to success or failure and to determine how roadblocks and inhibitors were overcome. Where possible, we will collect economic data to be used as input to our economic models.

In summary, we are focusing on issues of methodology, organizational structure, organizational behavior, and economics in software reuse. We expect to produce a set of recommended guidelines for reuse programs, incentives for reuse at various organizational levels, and some economic models of reuse.

Reuse: Where to Begin and Why¹

Robert Holibaugh

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
NET: rnh@sei.cmu.edu

Abstract

One of the main impediments to reuse realizing its potential for improving software productivity and reliability is the large up-front investment that must be made. The Software Engineering Institute (SEI) was interested in identifying the benefits of reuse to the MCCR community, and so the SEI was faced with the similar problem of how to investigate reuse without making a large up-front investment. This paper examines the general advantages and disadvantages of various starting points for reuse. Finally, we analyze the decision made by the SEI's Applications of Reusable Software Components Project.

1. Background

1.1. Introduction

The need for increased productivity and reliability are two of the challenges that led to the formation of the Software Engineering Institute (SEI). Meeting these challenges is a fundamental goal of software engineering, and these challenges are being addressed by the community at large, some DOD organizations, and academia. Furthermore, these problems are being addressed by using newer languages and techniques such as Ada and object-oriented design, by evaluating current development tools and environments, and by measuring and understanding the software development process. Work in these areas and comparisons with other engineering disciplines has led to renewed interest in the promise of software reusability for decreasing life-cycle costs and improving reliability.

Understanding the factors that make software artifacts (requirements, designs, code, test plans, and documentation) reusable and applying those resources successfully will help to transition this promising software development approach into common practice.

1.2. Potential for Reuse

The reuse of software resources has the potential to improve productivity by reducing or eliminating the cost of implementation, integration, documentation, and testing of code. The time saved by modifying reusable code to fit one's application will also reduce the total effort. Furthermore, if a set of reusable resources is part of a larger design, a significant amount of effort will be saved during design and in interfacing the set of reusable components. The time spent coordinating a design among the development staff may actually be the greatest overall savings from reuse.

In addition to increasing productivity, reusable resources can also increase the reliability of software by reducing the coding, interface, and documentation errors. Since reusable resources have been demonstrated to be reliable, the reliability of the system under development will be increased. Reusable resources also have the advantage that they have been used by other developers who are not constrained by the implicit assumptions and approach of the original developer. This use is a very effective type of testing which is somewhat similar to mutation testing. The inclusion of a reusable resource in each new development further tests the resource thereby increasing the probability of executing any given segment of code. When a set of components which implement a larger functionality are reused, they further

¹Sponsored by the U. S. Department of Defense

reduce interface and communication errors.

Domain specific software, which solves recurring problems in the domain, has the potential to improve the consistency and maintainability, and to reduce the time necessary to complete the system. Domain specific reuse implies that the components and their accompanying requirements and design are reused. This higher level reuse promotes standard designs for systems and subsystems. Thus, the consistency of systems from one generation to the next is increased, so maintainability of the systems is increased and the training time for new maintainers is decreased. The impact of personnel turnover, changing hardware, and other vagaries of software development are reduced by reusable designs which capture high level corporate expertise. The standard decomposition of compilers and their reusable tools, lex and yacc, are an example of this type of savings.

Software reuse also promotes maintainability because standard designs reduce learning time, and reusable artifacts should be simple and easily understood [MCNIC86]. When the development uses a standard design, more work can be done in parallel and the development schedule is shortened. Another benefit of standard designs is that the interfaces are well understood and work can proceed in parallel. Since many of the problems have been encountered before and are resolved in the design and code, the complexity of the problem is further reduced. Again, the techniques and knowledge typically taught in an undergraduate compiler course are a simple example of the advantages of reuse.

1.3. A Major Reuse Impediment

If reuse can so dramatically increase productivity and reliability, then why don't we see more reuse? The answer is that there are too few examples of successful reuse for government or industry to make the large up-front investment. A strategic approach would identify potential applications for reuse, do a domain analysis, develop reusable resources and tools, create a reuse-based software development methodology, and apply the reusable resources to improve productivity and reliability in future efforts in that domain. Therefore, a major impediment to more successful reuse is how one can demonstrate the practical advantages of reuse without a large up-front investment.

1.4. Large Initial Investment

The cost for domain analysis, which includes collecting, organizing, analyzing information about the domain and representing the results, would be high, but the cost for constructing reusable components and tools would be much higher. The effort for a domain analysis could easily be one staff year with no guarantee that common functions, objects, or other reusable resources would be identified. The Common Ada Missile Packages (CAMP) domain analysis required approximately 7.5 staff-months of effort [MCNIC88]. Current estimates for the cost of developing reusable software indicate reusable software require twenty percent more effort than non-reusable software [MCNIC86]. Furthermore, producing reusable designs and developing reusable subsystems will undoubtedly cost more than producing a single design or single subsystem. Since there are very few examples of reusable designs, the cost to achieve this level of reusability could be very large indeed. Guidelines [ST.DE86, GARGA86, SOFTE85, PRESS83, STARS86] for creating reusable code have been published, but we don't yet have empirical evidence that these guidelines will produce reusable code. Finally, the cost for creating a reusable subsystem of say 10,000 lines of code could exceed \$500,000. The CAMP project implemented 16,000 lines of code with slightly more than 5 staff years of effort [MCNIC88]. Without actually implementing a system with reusable resources, we really can't determine the effectiveness of the domain analysis and domain engineering. There is very little, if any, evidence that domain analysis and domain engineering produce effective reusable resources. Without criteria to determine the effectiveness of the domain analysis and domain engineering, this is a large initial investment with no means to evaluate the result.

2. The SEI and Reuse

2.1. Constraints on a Reuse Investigation

Because the SEI agreed with the recommendations made by the Defense Science Board on the potential productivity payoffs of reuse, the SEI became interested in examining the costs and benefits of reuse, in demonstrating a proof of concept, and in acquiring and transitioning state-of-the-art reuse technology.

investment necessary for domain analysis seems a very risky proposition.

The advantages and disadvantages of a domain analysis seem reasonably well balanced in the abstract, so each organization must evaluate the risk and benefit to its bottom line. There are some examples of the benefits of reuse in the data processing industry. [LANER84] The Japanese have also made a long term investment in reuse and been reasonably successful. [MCNAM86] These examples are not MCCR systems, but it does indicate reuse has real potential for MCCR Systems. Domain analysis is the first step in constructing reusable resources, and more of these efforts are being initiated by government and industry, but the results are not yet available. As we gradually overcome the lack of domain analysis experience, and as the importance of capturing domain expertise is recognized, more domain analyses will be done, and we will be able to justify the up-front cost. The large up-front cost may always remain, and can only be justified by making a long term business commitment to the domain. As time goes on, the advantages of starting with domain analysis begin to outweigh the disadvantages.

3.2. Component Construction

The potential disadvantages of initiating a reuse effort with component construction without a domain analysis outweigh the potential advantages. The risk of making a large up-front investment in component construction without the benefits of a domain analysis far outweighs the potential advantages, which apply mainly to a product based organization. The disadvantages, however, apply to SEI as well as product based groups.

One advantage of beginning a reuse effort with component construction is that the organization already has the capability to develop domain specific code. Organizations which develop software for a specific market like radar systems possess the expertise to develop radar support software. The work done in domain engineering is very similar to the work done in the design and coding phases of standard projects. Unlike domain analysis where an organization lacks expertise, they do possess the expertise to develop detailed designs and code.

Another advantage is that this effort builds assets for the future. Like training or investments in work

stations, reusable resources are fixed assets. The output of this component construction is concrete, so that it can be measured and at least examined qualitatively by the organization's experts. Furthermore, it can be compared to current and previous developments to determine how adequately it solves current and past problems. Once again, the advantages of initiating a reuse effort with component construction are strongly related to needs and expertise available in product based groups.

There are several significant disadvantages to initiating a reuse effort with component construction. The main technical problem is knowing what to build. This problem is similar to developing a system without having a set of requirements. In today's resource constrained world, starting development without a set of requirements is extremely risky. Once again, a very large up-front investment is required, so the investment in component construction could be 5 to 10 times the cost of a domain analysis. [MCNIC88] Furthermore, we don't have guidelines or validation techniques for the reusable resources. Finally, the construction of large-grained reusable components or tools to generate large-grained components will cost even more to develop. Initiating a reuse effort with components construction require a very large up-front investment and has considerable risk because of our inability to validate the results.

The advantages and disadvantages of initiating a reuse effort with component construction aren't well balanced in the abstract, so we do not recommend starting a reuse effort with component construction. After a domain analysis such an investment seems quite reasonable and is absolute necessary for reuse. An organization could, however, construct the reusable components as part of an existing development, basing their construction on the results of a domain analysis. In any case, component construction should only occur after a domain analysis.

3.3. Library Construction

The disadvantages of initiating a reuse effort with library construction appear to outweigh the advantages. The main advantage is that we all understand the necessity of constructing a library for reusable resources, but if we don't have resources to store in the library, we won't be able to recover the cost without further investment in domain analysis and domain

engineering. The ability or inability to recover one's investment is the most significant factor in deciding to initiate a library construction project.

The advantage to library construction is its reasonable cost, which appeals to management. The construction of a library for a single division or company may not require more than one to two staff years of effort. Since the library could apply to all domains and all projects within the company, the cost doesn't seem excessive. In fact, it is not unreasonable with appropriate planning for the library to be useful to several divisions of a large aerospace corporation. The key advantage to library construction is that it is absolutely necessary for the ultimate success of reuse. The storage, control, and configuration management of the requirements, designs, code, documentation, and their history is absolutely necessary for the developers who will apply reusable resources. Finally, management does not have a problem understanding the need for a library, and they easily recognize its applicability to several projects.

An organization that constructs a library before they have identified what they will store in the library may never realize any benefit from the library. The cost of domain analysis and domain engineering may prevent management from allocating the resources to construct the reusable resources. The classification techniques which are most applicable to one domain may not be applicable to other domains. Unless the library can support multiple classification mechanisms, it may not be able to support resources which are constructed after the library is implemented. In fact, some of the security restrictions on classified or sensitive resources may prevent users from accessing the library. Even though there are several libraries under construction, there is little in the literature on guidelines and procedures for library operation.

The recovery of library implementation costs will depend on the availability, quality, and applicability of the resources stored in the library. Some organizations plan to place all software that they construct into the library. This may create a serious classification and retrieval problem because the volume of resources can create more confusion, and there may be very little that's reusable. The disadvantages of starting a reuse effort with library construction are primarily related to recovering the cost which is related to being able to retrieve resources that are yet to be built.

The disadvantages of starting a reuse effort with

library construction outweigh the potential advantages since the developer may never be able to recover the cost without making an even larger investment. After a domain analysis and domain engineering of reusable resources, library construction has much less risk. In fact, the classification of the resources which defines the retrieval mechanism is derived from the domain analysis. Library construction before domain analysis is difficult to justify unless one already has a large collection of reusable resources.

3.4. System Construction

The disadvantages of initiating a reuse program by constructing a system from reusable resources far outweigh the advantages. There is great risk in developing a system from reusable resources without establishing the applicability, understandability, and quality of reusable resources. This risk is definitely unacceptable except for shadow or R&D projects.

The purpose of initiating a reuse project in system construction is to validate the cost and benefits of reuse, investigate reuse technology, and provide a reference point for future reuse. The costs and benefits of reuse have not been determined empirically, and there are very few examples of successful third party reuse. By starting with system construction, these costs and benefits can be determined by collecting data on the development. An organization can also investigate the effectiveness of the reuse technology which was used to supply the reusable resources. This information is very important for constructing additional reusable resources. A project that constructs a system from reusable resources is certain to encounter unexpected problems and to gain insight into the solution of some expected problems. This information is very useful for construction of additional reusable resources and for planning and scheduling other reuse development efforts. The advantages of initiating a system development with reusable resources can be grouped into gaining practical experience with new technology.

The disadvantage of starting with system construction is the risk inherent in using new and unproven technology. The risk to real projects of using third party software can be reduced by prototyping and testing the components, but this will certainly reduce the productivity gain. So, it seems that nothing is really gained without assuming the trustworthiness of the resources. Does this mean we are trading reliability for

productivity? Any gain from productivity cannot offset problems in reliability. The gains expected from reuse can only be assured *if* we have reliable resources. Where can one obtain the reliable resources? Abstract data types (ADT) can be purchased commercially, and they are guaranteed, but the gains from these ADTs will have to be amortized over many developments. To show significant productivity gains on a few projects, the resources probably need to be domain specific. With the exception of the CAMP components, there may not be another rich and powerful set of reusable components. The main disadvantage of building a system with reusable resources is the risk of the technology. That is assuming the reusable resources exist at all.

In general, the disadvantages of beginning a reuse effort with system construction outweigh the advantages, and we would not recommend it. The lack of reusable resources and inherent risk to the project far outweigh the potential advantages. If an organization could acquire an effective set of reusable resources, and if the failure of the project to deliver a system is acceptable, then this may be a viable starting point to investigate reuse.

4. Conclusion

In the case of the SEI, we could overcome the disadvantages of starting with system construction. We acquired the CAMP components, and we initiated a project to redevelop one of the ten missiles from the CAMP domain analysis. This did not, however, solve all of our problems. We didn't have the necessary missile expertise, and we didn't have the means to test the software that was constructed. Raytheon provided the missile expertise, and the Cruise Missile Program Office provided the Interpretive Simulation Program for the testing the final software. So, we initiated a redevelopment of a Tomahawk missile. From that development, we learned several important lessons which we will report to the community later this summer.

5. Bibliography

GARGA86 Gargaro, Anthony and Pappas, Frank. *Ada Reusability Study*, Computer Science Corporation, Moorestown, New Jersey, August 1986.

- LANER84 Lanergan, R and Grasso, C. Software Engineering with Reusable Designs and Code, *IEEE Transactions on Software Engineering*, 10(5): 498-501, September, 1984.
- MCCAI86 McCain, Ron. *Reusable Software Component Engineering*, IBM, Houston, Texas, July 1986.
- MCNAM86 McNamara, Don. "Japanese Software Factories", in Proceeding of the Software Factory Forum, SEI, February, 1986.
- MCNIC86 McNicholls, D., et al. *Common Ada Missile Program*, McDonnell Douglas Astronautics, St. Louis, MO, Air Force Armament Laboratory, Eglin AFB, Florida, AFATL-TR-85-93, May, 1986.
- MCNIC88 McNicholls, D, et al. *Common Ada Missile Packages - Phase 2*, McDonnell Douglas Astronautics, St. Louis, MO, Air Force Armament Laboratory, Eglin AFB, Florida, AFATL-TR-88-62, November, 1988.
- PRESS83 Presson, Ed, Tsai, J., Bowen, T., Post, J., and Schmidt, R. *Software Interoperability and Reusability Guidebook for Software Quality Measurement*, Boeing Aerospace Co., Seattle, Washington, Rome Air Development Center, Griffiss AFB, New York, RADC-TR-83-174, July 1983.
- SOFTE85 *Ada Reusability Guidelines*, Softech, Incorporated, Waltham, Massachusetts, April 1985.
- STARS86 "STARS Reusability Guidebook V4.0," STARS Application Workshop, NRL, San Diego, California, September 1986.
- ST.DE86 St. Dennis, R. *A Guidebook for Writing Reusable Source Code in Ada*, Honeywell Computer Science Center, Golden Valley, Minnesota, May 1986.

POSITION PAPER ON SOFTWARE REUSE

Dr. Harry F. Joiner
Telos Federal Systems
55 N. Gilbert St.
Shrewsbury, NJ 07702

The issue of a practical application of reusable software has been divided into six subissues:

1. Definition of requirements with reuse as an objective
2. Design of software with reusable components
3. Issues of reuse with and on existing systems
4. Assisting the programmer with reuse
5. Creating incentives for reuse
6. The Ada language role in reuse

As this outline indicates, much of the life cycle will be effected by the adoption of a reuse approach to software engineering. Just as education and practice in the hardware part of the electronics industry have changed dramatically since the days of individually designed components, the training and procedures of software engineering will change significantly as reuse of software components becomes common practice.

1. Definition of requirements with reuse as an objective

The user requirements definition and analysis should be independent of implementation to the greatest extent possible, leaving the maximum flexibility available for the engineers to determine the hardware and software solutions. Furthermore, system and software requirements definition should not be limited by the availability of reusable software components any more than it is by predesigned hardware components. Software requirements should, however, take into account the effects of reusability on the design approach.

A valid objective that can be stated in the requirements phase is to maximize reuse of software. Having this objective clearly stated will be an important incentive during the earlier years of applying reusable software. Another valid objective is to add to the reusable components available in libraries.

2. Design of software with reusable components

Designing with reusable software components will place a different emphasis on the software engineering effort. The use of predefined components places greater importance on optimal use of the engineer's toolbox and less emphasis on detailed understanding of computer languages and programming techniques. Two examples of current practices that reflect this difference are seismic processing in the oil industry and hardware design.

Seismic processing includes sophisticated Digital Signal Processing (DSP) techniques applied to large data sets (tens of thousands of time series with several thousand samples each). The DSP is performed by using a software package which routinely contains over 100 components, each executing one or more tasks.

such as band pass filtering, sorting, wavelet shaping, or other static or dynamic corrections to the data set. The necessary parameters and the appropriate data set are passed to each component or module as required. The geophysicist (seismic processor) may generally choose the order of processing and the components to be used along with the specific parameters to be used for each process. The processing sequence(s) will vary depending on the quality of the data, the objectives of the survey, and form of the presentation. The whole process can be properly viewed as a high level design with reusable components. Some of the modules will have been in the inventory since the mid-60's while others will have been incorporated very recently in this rapidly developing area of signal processing. A new geophysicist will require a period of months to become familiar with the processing package in order to take advantage of the variations and choices in components for filtering, migration, scaling, and other functions. At present most of the required information for this is contained the user's manual for the seismic package. The experienced processor will be able to familiarize him/herself with a new system or package in a much shorter time because of his/her knowledge of other similar packages. The basic process is simply one of designing a specialized DSP program using reusable software components.

The second example is probably a more familiar one: designing hardware from the enormous inventory of electronic parts and processing chips available from today's manufacturers. Only a very small percentage of a hardware system today is created from scratch, and the hardware engineers manage to obtain the required specifications, test data, etc. to accomplish the task from vendor information, catalogs, and other sources.

A critical element in the design from reusable components approach is that the design may accomplish its objectives in a different manner and that the engineer should be trained to view the problem from this different perspective.

3. Issues of reuse with and on existing systems

A critical issue relating to the millions of lines of existing code is how to determine what code is suitable for reuse and how to approach this task in a cost-effective manner. Some current programs, such as the Army Tactical Command and Control System (ATCCS), are designed to provide a source of reusable components, but guidance for which components are to be reused and how they are to be certified or qualified for reuse has not been defined.

The ATCCS example illustrates a large, domain-specific system that should be able to take significant advantage of reuse in fire support functions, battlefield planning, maneuver control, logistics, and systems operations and interfaces. Reuse is being designed into the system architecture with the adoption of the Common Hardware/Software system that should insure reuse of many of the support, communications, and system software components. When it comes to the applications software, the situation is somewhat different. Two of the five ATCCS components (AFATDS and MC) are well into the development programs without clear guide-

lines on what qualifies for reuse or should be designed for reuse.

4. Assisting the programmer with reuse

A major prerequisite for widespread reuse of software components is to obtain the active cooperation of the designer/programmer. The principal requirements for this cooperation are:

- o Ease of accessing the reusable components
- o Knowledge of the requirements to make the components work in the new situation
- o Confidence that the reused component will not cause problems

As indicated by the preceding examples, these can be overcome with current technology and adequate training in the use of individual reuse libraries. Although much more can be accomplished with databases, fully commented specifications, and information regarding algorithms used, test results, interface requirements, and limits of operation, a willingness to adapt design and coding practices to maximize reuse of code from corporate and public reuse libraries will go far.

Training in college and on-the-job in how to design for reuse, in familiarizing the software engineer with specific reuse libraries, and in coding practices that include interfacing with and using the code of others will develop the skills and attitudes needed to take advantage of reuse libraries as they are created. Computer science majors are encouraged to believe that writing code is the ultimate function of system design and development, placing the emphasis on solution techniques that are designed for small systems and problems. This approach also encourages a pride in code authorship that makes it difficult to accept the requirements for reuse of someone else's code on large systems. Proper training in the advantages and techniques of reuse can eliminate these problems, just as they have in the transition from design from scratch to using off-the-shelf components in the hardware field.

In order to perform adequate training in reuse, colleges and universities must have access to one or more significant reuse libraries. Students who become accustomed to operating with these reusable components will carry over the habits and familiarity with the software to their work environments.

5. Creating incentives for reuse

While training and the accumulation of reuse libraries will provide some incentive to the aggressive software engineer, real progress will be made only by creating significant incentives and reducing the true stumbling blocks to reuse in practice. These changes need to occur at two levels: those that effect corporate external relations and those that involve the internal management

of software projects. The changes required are mostly political and "cultural" in nature, not technical.

The first category of incentive (corporate external incentives) must address corporate profitability and liability. Reuse will become a significant factor in the near future (5-15 years) due to its long-term savings and the increased competition for software development work. In several areas, such as COBOL financial packages and the Japanese software factories, reuse libraries have become common practice and the only way to stay competitive. They are limited now to certain restricted applications or hardware/software environments, but will become increasingly widespread.

Because no significant application of reuse is currently being made there and reuse setup costs time and money, corporate profitability in the Government sector is negatively affected for reuse. The disincentives abound in the cost-plus development contracts that encourage reinventing everything from the wheel to the ballistics software package. Firm Fixed Price contracts are even worse because the development of reusable code is more costly, even if the long-term benefits are enormous. Government incentives through extra awards for contributions to or applications of reuse could be provided as value engineering similar to the hardware program. Numerous Government programs are discussing the development of reuse libraries, but none have incentivized these projects for the corporation. Testing, documentation, and generalization all add to the expense and usefulness of library components and must be supported.

The related issues of ownership and liability for components in the Government reuse libraries have not been addressed either. A royalty arrangement might provide real incentives for corporations to contribute to reuse libraries, but determination of the limits of liability must be in place at the same time. The profit and liability must balance in a reasonable way.

When the profit motive strongly supports reuse, the second issue of internal management support will be incentivized. Significant support for reuse by the team leaders, supervisors, and managers of the company will enforce reuse by the designers/programmers. Code reviews with specific attention to reusable components, reuse competitions between project teams for contributions to and applications of reuse, and management commitment to the extra costs of developing and maintaining reuse libraries are all important ways to implement a corporate reuse strategy.

6. The Ada language role in reuse

Ada supports reuse through the implementation of packages and generics, including particular support for information hiding and abstraction. The separation of specifications from bodies in the package construct encourages full commenting in the specification for the potential user of the package while controlling access by the user to the actual implementation in the body. Ada incorporates reuse in the language definition through the use of predefined packages (TEXT_IO, e.g.).

The commenting of the specifications could easily provide a mechanism for supplying the potential user with his/her required information, including test data, algorithm specification, limitations on range of use, and interface requirements.

7. Reasons for attendance

I have 12 years experience in seismic processing, technical studies and modeling (DSP), and project management. The past three years, I have worked on Government projects, primarily in Ada, reviewing management and technical issues. My current responsibilities include support for the Advanced Field Artillery Tactical Data Systems (AFATDS), part of ATCCS, in the areas of project management, software metrics, design methodology, and software reuse.

The industry will accomplish the required 3-5 fold increase in productivity over next decade or so only through the active implementation of reusable software components, designs, and templates. My experience indicates it can be done and, to remain competitive, it must be done.

MAXIMIZING ADA REUSABILITY

Reusability has become the latest "buzz-word" of the Ada community. It has many different conflicting definitions. These definitions run the gamut from including any design or code used more than once to only code used on two or more widely dissimilar projects. The community also seems to be in violent disagreement as to whether reusability exists or is just a myth. Some of the statements have been; "of all the code written in 1983, probably less than 15% is unique", "very little research being conducted ... in Ada Reusability", "40 to 60 percent of the code was repeated", "reuse factors of 85% have been reported", "no credible methodology...to provide the reuse of source code between widely dissimilar application areas", "68 to 95% of existing software (was reused)", "no reliable method of storing or retrieving items".

The first step in maximizing reusability is to define it. Since one major purpose of reusing software is to reduce the effort needed to produce a program and another is to reduce the cost of maintenance, the most liberal definition should be used. Therefor reusability could be applied to code, design, or requirements definition that can be used more than once. The real key to maximizing reusability is to start with the requirements definition.

Requirements definition must be performed in a manner that will promote reusability in software design and coding. Object Oriented methodologies seem to be gaining favor in major Ada projects. Therefor an Object Oriented requirements definition methodology should be carefully employed in order to promote reusability.

The impact of not laying a firm foundation for reusability during the requirements analysis phase of a program can have disastrous results. An example of this can be found in a large Ada project started in 1984. The original contract was awarded for 33 months (200 - 300 KLOC). After 4 months the customer made an assessment that insufficient progress was being made on the contract.

After 7 months the contractor informally notified the customer of a potential growth in the contract. The causes of the growth were failure to fully comprehend the scope of the user requirement for automation, overestimation of the contractors ability to write software in a new language (Ada), bid errors, and misinterpretation of solicitation requirement for quality of product for test.

After 9 months the customer determined that the reasons for program growth were; contractor underbidding, an increase in the lines of codes, and no requirement for a functional definition. After 37 months the contract was extended to 48 months. The contract was eventually completed after 58 months (over 400 KLOC).

It was determined that one of the major reasons for the program growth was the inability of the contractor to effectively transition from a set of functional requirements to an object oriented methodology. This resulted in a much lower reuse factor. The contractor did manage to have approximately 25% of the code in a common library. However, during the last 6 months of the development the code size shrunk. This was due to the fact that the contractor had discovered many cases of duplicative code. It has been estimated that over 50% of the code could have been shared if the contractor had started with an object oriented methodology during the requirements analysis phase.

The proposed object oriented requirements analysis methodology is based on the identification and description of objects, states of the objects, and processes that transform an object from one state to another. The first step is to structure the requirements definition activity into several stages; object identification, object description, object states identification, object states description, process identification, and process description.

Maximizing reusability starts with the identification and naming of the objects. A generic method that does not limit the design or implementation should be used. This is also the stage of development in which the initial structure of the reusability library retrieval system is determined. Classes and subclasses of objects may have to be identified in very large and complex systems. The retrieval system should be structured so that it doesn't become overly complicated.

Providing object descriptions that support and promote reusability is very difficult. A dictionary of acceptable and generic keywords and phrases has to be established. The description of each object should contain its attributes. The attributes of different objects have to be compared in order to identify commonality of terms. The attributes should also be analyzed in order minimize the semantic differences.

The states that an object can exist in must be identified next. The states of the different objects should be compared in order to identify additional objects that belong to the same class. Care must be taken to avoid semantic problems. The object state descriptions can help if they are written using a consistent methodology.

The final and most critical step is to identify the processes that transport an object from one state to another. The process description is key to obtaining a clear identification of equal or similar processes. These processes will drive the design and code directly. A library of processes with a powerful identification and retrieval capability should be set up.

The above methodology for implementing an object oriented requirements analysis approach will serve as a firm foundation for programs that maximize reusability. Although it maximizes reusability within the project it would also serve as a basis for sharing design or code reusable components between projects.

However, reusability will never be implemented to any significant degree until software purchasers provide enough incentives to the software developers. These incentives must result in increased business and profits for the developer that reuses software if they are to be effective.

Stanley H. Levine

Technical Management Chief for
the Project Manager of Field
Artillery Tactical Data
Systems

Coming to Terms with Terminology for Software Reuse¹

A. Spencer Peterson

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
NET: asp@sei.cmu.edu

Abstract

There are problems with the use of many of the terms used in software engineering when applied specifically to reuse. Three terms of particular interest, taxonomy, software reuse, and domain analysis and some problems with their usage are discussed. The specific problems with these terms are generalized and several solutions are given, the most important being the introduction of the concept of a reuse process model to provide context and an overall view of the potential areas of discourse in reuse. Several new terms are proposed for future use as well as definitions that are meaningful in the context of software reuse.

1. Introduction

It is not possible to have an intelligent discussion or write a technically meaningful paper without a vocabulary to describe the essence of the thoughts the speaker or writer is attempting to convey to the listener or reader. That vocabulary must have two properties. First, it must be broad enough to describe the wide range of ideas that a user may wish to convey. There must be enough terms to describe the subject area. Second, the terms in the vocabulary to be used must be meaningful to both parties. The terms must have definitions. Furthermore, for use in technical discourses, the definitions must be few and precise enough to be unambiguous.

The English language now has over 500,000 terms with definitions totalling 59,000,000 words, according to the latest version of The Oxford English Dictionary (OED) just released [OXFOR89]. Even this number of terms is inadequate for the breadth of discourse in all

areas using English as the media for the capture and exchange of information, as new terms constantly come into being. English, overall, is inherently ambiguous due to the large number of meanings for many words in the language. The OED carries 200+ definitions for the word "set" used as a verb and over 50 more when one includes its usage as a noun. Even after noting that the OED carries many obsolete definitions for the purpose of tracing the history of word usage, it is no wonder why people writing natural language processing programs are having difficulties.

Technical jargon suffers even more from the problem of ambiguity because terms can pick up a special meaning within individual organizations that is precise and meaningful within them but ambiguous and confusing to outsiders. Software reuse has many problems to solve before it becomes common practice in the software engineering community. One problem is that there is no standard definition for the term "software reuse" itself! How can we say we practice software reuse when we (experts!?) don't agree on what it means, much less the community at large?

2. Problems with "Standard" Terms

Three often used terms illustrate the problem with the vocabulary for reuse. The term "taxonomy" is misused by many authors to mean the list of terms produced by doing a classification for a collection of objects. This usage is incorrect. A *taxonomy* is the process and procedures for creating a classification for a group of objects and is formulated and described in terms useful to those doing classifications. Biologists call a group of organisms that fits one of the categories of the formal

¹Sponsored by the U. S. Department of Defense.

classification units they use, such as phylum, genus or species, a *taxon*. We can take this word and create a new definition that fits a need in our discussions. Section 5 contains a glossary for the new terms and modified definitions introduced in this paper and are printed in italics for easy identification.

The word 'software' in the term 'software reuse' is not adequately defined to provide any meaning to the word that is proper for a useful definition of software reuse. Software is defined in [IEEE88] as "Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system." Do we look at reusing entire computer programs? The current state of the practice in software reuse is focused on reusing pieces of computer programs, in particular code components. Thus, it is necessary to extend the definition of 'software' in a widely used standard glossary or dictionary to get a meaning that is adequate for use in OUR discourses on 'software reuse'. Section 5 contains a proposed definition for *software* and *software reuse*.

The term 'domain analysis' suffers from a similar problem. The word 'domain' does not have a definition that is usable when one attempts to analyze the two words in the term. The only useful definition in several dictionaries is derived from mathematics and the word "set", a collection of elements, is used. But the software engineering community talks about things such as abstract data types, communications, missiles, control systems as examples of domains. These are things where code components and/or whole systems are developed for use on computers. But this doesn't provide enough information to lead to a good definition. So we must try to analyze what we mean by the term 'domain analysis' to derive a usable definition for 'domain'.

To do such an analysis, we must try to envision where domain analysis fits into the overall process of reuse and the software lifecycle and determine its inputs and outputs, like the models we derive during systems development. What information are we trying to capture when we do 'domain analysis'? We want to extract the commonality and differences in the various components and systems developed that achieve the desired capabilities in the area of interest. With this and the answers to other such insightful questions, it is possible to come up with a definition for *domain* that is useful, both by itself and when we create compound terms such as *domain analysis*, *domain engineering*,

and *domain model*, all used in Section 4 and defined in Section 5.

3. What can be done to solve the Problem

What have we seen in the three cases of attempting to define terms that are given above? First, we must be careful not to abuse or misuse terms that already have a precise and useful definition in other contexts or areas, or at least admit to the non-standard usage of a term and provide the reader with the definition of the term as it is used in later discourse.

Second, we must agree that some terms used in software engineering are inadequately or inappropriately defined for use in the discussion of reuse, to change their definitions as necessary, and get the changes included in standard glossaries and dictionaries used in our field. Only after we agree upon a set of terms and definitions that fits our needs and get them published in a widespread way can authors write more precisely and readers understand the intended meaning.

Third, and most important, the first step towards the solutions listed above is: we need a model for the process and activities that play a part in reuse. A model provides the context for using terms and for naming and describing processes such that the meanings are well understood. Such a model must be abstract enough to not be overly constraining and complete enough to present a fairly comprehensive approach to understanding the many facets of software reuse. Such a model is considered in the following section.

4. A Process Model for Reuse

The model depicted in Figure 4-1 on the next page is not to be considered as the only approach to a process model for reuse, but it illustrates the benefits of having a model and, in many ways, it reflects the SEI Software Reuse Project's view of reuse. It is not possible to provide anything close to a comprehensive description of the model as such a description would be the subject of an entire paper in and of itself. The description following the figure provides enough information to give context and meaning to many of the terms and definitions given later in Section 5.

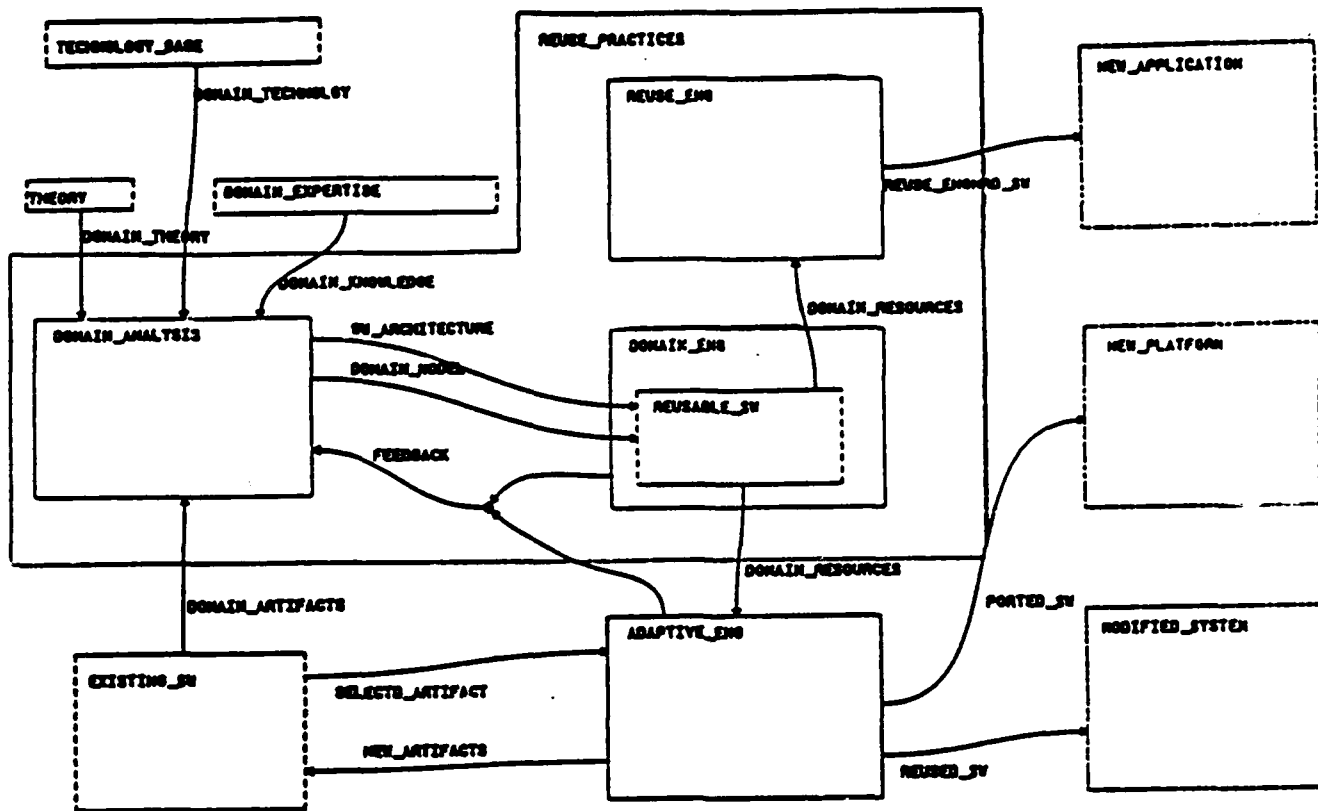


Figure 4-1: Process Model Incorporating New Reuse Practices

First, note the area enclosed by the polygon labeled *Reuse_Practices*. It surrounds a set of activities that ought to be performed by most software engineering organizations that do recurring work in a *domain*. *Artifacts* from previous developments are extracted from an existing base of software, and with this data and other information such as underlying theory, current and emerging relevant technologies, and the knowledge of personnel experienced in the domain, a *domain analysis* is performed. Its outputs are primarily a *domain model* and a *software architecture*. *Domain engineering* is performed to build and control a library of *reusable software resources*. Projects working on applications in the domain of interest can draw upon those *resources* while performing *reuse engineering* to produce a new application in the domain. Feedback as to the effectiveness of the resources, suggested modifications, etc. is given so that the domain data can be updated as appropriate.

The bottom area of the figure illustrates the type of *reuse* that is actually practiced by some groups doing substantial software development work. The existing software base is searched via some mechanism for an

artifact that is potentially reusable. *Adaptive engineering* is performed to modify the artifact for use in a modified system or on a new platform. This process can be improved by implementing the new reuse activities given above. Domain engineering provides a better mechanism for more successfully locating candidate resources for reuse, and feedback can ensure that missing resources or those with problems are created or modified to fit ongoing needs.

Note the use of the two terms *artifact* and *resource*. Some would claim they are the same. We believe that some artifacts may be resources, but most are not because they do not have a high degree of *reusability*. Any development process results in artifacts, but few can deliver good resources. A good resource for reuse must have multiple pieces that relate to one another. These pieces provide useful information that is applicable at different phases of the development process. This entity with related reusable resources is an *asset* and is the basis for developing a good *software library*.

5. Derived Terms and Definitions(A Partial Glossary for Reuse)

The terms and definitions are taken from a draft update to ANSI/IEEE Std 729 (Glossary of SW Terminology) [IEEE88], except where the term is marked with an (M) for Modified where inserted text is enclosed in [], or with a (*) signifying a term that is not defined in the IEEE draft. Other comments are enclosed by {} and placed at the end of the definition.

abstract data type.

A data type for which only the properties of the data and the operations to be performed on the data are specified, without concern for how the data will be represented or how the operations will be implemented.

abstraction.

(1) A view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information. (2) The process of formulating a view as in (1).

adaptation data(M).

Data used to adapt a program [or component] to a given installation site or to given conditions in its operational environment.

adaptation parameter(M).

A variable [or placeholder] that is given a value [or other appropriate information] to adapt a program [or component] to a given installation site or to given conditions in its operational environment.

adaptive engineering(*).

The process of modifying a system or component to perform its functions in a different manner or on different data than was originally intended.

adaptive maintenance(M).

Software maintenance performed to make a computer program [or component] usable in a changed environment.

application-oriented language.

A computer language with facilities or notations applicable primarily to a single application area. (This fits the Neighbors concept of DOMAIN LANGUAGES.)

architecture.

The organizational structure of a system or component.

artifact(*).

Any product of the software development process.

asset(*).

A set of reusable resources that are related by virtue of being the inputs to various stages of the software life-cycle, including requirements, design, code, test cases, documentation, etc.

(Note: an asset can be a design and the control code for using other assets in the library in a more powerful way. Assets are the fundamental element in a reusable software library.)

component.

One of the parts that make up a system. (Note: a component is some useful portion of a computer program. It may be subdivided into other components.)

control abstraction(*).

(1) The process of extracting the essential characteristics of control by defining abstract mechanisms and their associated characteristics while disregarding low-level details and the entities to be controlled. (2) The result of the process in (1).

data abstraction.

(1) The process of extracting the essential characteristics of data by defining data types and their associated functional characteristics and disregarding representational details. (2) The result of the process in (1).

domain(*).

The set of current and future systems/subsystems marked by a set of common capabilities and data.

domain analysis(*).

(1) The process of identifying, collecting, organizing, analyzing, and representing a domain model and software architecture from the study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest. (2) The result of the process in (1).

domain engineering(*).

The construction of components, methods, and tools and their supporting documentation to solve the problems of system/subsystem development by the application of the knowledge in the domain model and software architecture.

domain model(*).

A definition of the functions, objects, data, and relationships in a domain.

functional abstraction(*).

(1) The process of extracting the essential characteristics of desired functionality by defining it abstractly along with its associated behavioral characteristics and disregarding low-level details. (2) The result of the process in (1).

master library.

A software library containing master copies of software and documentation from which working copies can be made for distribution and use. (This should be meticulously maintained and controlled by a special group of reuse engineers and librarians.)

modularity(M).

The degree to which a system, computer program [or code component] is composed of discrete components such that a change to one component has minimal impact on other components.

perfective maintenance(M).

Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program [or component].

production library.

A software library containing software approved for current operational use.

resource(*).

Any software entity placed into a software library for purposes of reuse.

retirement(M).

(1) Permanent removal of a system, component [, or resource] from its operational environment [or the master library.] (2) Removal of support from a operational system, component [, or resource].

reusability(M).

The degree to which [a] software [resource] can be used in more than one computer program [or system, or in building other components or parts.]

reusable software(*).

Software designed and implemented for the specific purpose of being reused.

reuse(*).

The application of existing solutions to the problems of systems development.

reuse engineering(*).

(1) The application of a disciplined, systematic, quantifiable approach to the development, operation and maintenance of software where reuse is a primary consideration in the approach. (2) The study of approaches as in (1). (The same definition as for 'software engineering' given in the IEEE standard except for the addition of the phrase beginning with "where".)

software(M).

Computer programs, [code components and other artifacts], procedures, and possibly associated documentation and data pertaining to the operation of a computer system [or its components].

software architecture(*).

The packaging of functions and objects, their interfaces, and control to implement applications in a domain.

software library(M).

A controlled collection of software [resources] and related documentation designed to aid in software development, use, [reuse], or maintenance.

software repository.

A software library providing permanent, archival storage for software and related documentation. (The key word is 'archival'. Also note the word 'control' is not mentioned.)

software reuse(*).

(1) The process of implementing new software systems and components from pre-existing software. (2) The results of the process in (1).

specification(M).

A document [or other media] that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component, and, often, the procedures for determining whether or not these provisions have been satisfied.

taxon(*).

A group of resources constituting one of the categories in a taxonomic classification for reusable software in one or more domains. (The plural is taxa.) (Note: The American Heritage Dictionary's (AHD) only definition is cast in a biological context.)

taxonomy.

The science, laws, or principles of classification [AMERI85].

6. Conclusion

The problem with the terms and definitions used in software engineering that are of importance to discussions on reuse has been examined and several solutions offered. The most important of these is the concept of a model to provide a foundation for discussion and context for an examination of appropriate changes to existing terms and definitions. The model also provides a method for creating new terms with good definitions that will have meaning when used in future work in software reuse.

7. Bibliography

- [AMERI85] The American Heritage Dictionary, Second College Edition, Houghton Mifflin, Boston, MA, 1985.
 [IEEE88] Radatz, J. et al., Draft Glossary of Software Engineering Terminology (Update to ANSI/IEEE Std 729-1983), Institute of Electrical and Electronics Engineers (in review), August 15, 1988.
 [OXFOR89] The Oxford English Dictionary: Second Edition (Simpson et al., editors), Tarendon, Oxford, England, 1989.

REUSE EXPERIENCES/ENHANCEMENTS - A WHITEPAPER
REUSE IN PRACTICE WORKSHOP
SEI, Pittsburgh, Pa., July 11-13, 1989

Edward W. Beaver

Revised June 15, 1989

Copyright 1989 Westinghouse ESG, Baltimore Md. 21203

1. BACKGROUND

Reuse is initially defined by the industry as reapplying the same software (modified or unmodified) to a different system.

The industry has a divergence of opinion as to the scope of the reuse problem. One end of the spectrum believes that reuse is a major technical challenge to facilitate "automatic recomposition" of software on reapplication to a similar system. The other end of the spectrum advocates that reuse is simply a management problem of managing the library of available software and making it readily available to engineers for reuse. This paper describes the author's personal experiences and judgements on the software reuse issue.

The domain of a software item is the scope of its functional performance for a particular type of product in a particular environment. In that sense, it is analogous to the analysis made of a product's features to address a particular system problem. Thus the scope of the domain analysis can proceed from the system problem and then characterize the software product features and environments. Engineers who conduct domain analysis must be knowledgeable in the potential scope of all three of these items.

Westinghouse ESG Aerospace Software Engineering Department's experience in reuse dates from 1978, when portions of an F-4 Weapon System Operational Software, design, algorithms, and documentation were reapplied to other F-4 programs. Other 1980s experience has experimentally reapplied test systems and portions of mode software between radars. In other domains, proposal, mode design documents (MDDs), and software documentation materials have been "cut-and-pasted" as useful between proposals and projects. In each of these experiences, the reuse was done manually, based on an individual engineer's knowledge of the application and the availability of the existing materials, as well as a common environment that allowed an easy reapplication of the materials.

Reuse should be defined as reapplying any portion of the *systems or software engineering technology* applicable to the definition, design, development, and test of a system and/or software product in a similar domain. Automatic Reuse can be developed by assessing the elements present in manual reuse and defining how an automated browser or composer might use them.

EWB-REE061589-1/8

2. ELEMENTS OF SOFTWARE REUSEABILITY

Reuse is only valuable if the problem domain, possible solutions, and environments are compatible or similar. Reviewing and analyzing the problem domain, the potential solutions, and the environments is defined as domain analysis.

Reuse applies if the domain analysis [1] yields similarities in the problem domain of system functions, system interfaces, operating concepts, and operations intrinsic to the processing of functions, [2] determines that existing solutions are sufficient for the needs of the system, and [3] finds compatible or adaptable environments for the components or units.

Problem domains for embedded real-time mission-critical systems follow the system technologies themselves: airborne fire-control radar, airborne early warning (AEW) radar, ground based radar, shipboard radar, electronic warfare (EW), electro-optical (EO) systems, communications and navigation (CNI), command and control (CC), weapon control, mission control, telemetry processing, ESM/intelligence, etc. Each problem domain itself is subject to the functional scope of the hardware and the technology level. Further technology developments should also be anticipated as new problem domain requirements.

3. MULTIPLE LEVELS OF REUSE

Within the broad definition of reuse, seven levels can occur for a problem domain and solutions as shown in Table 1. Each level can potentially be reused. Specifications are treated as Sub_Configuration_Items (SCI's) with Specification_Components (SPC's) comprised of Description_Code (DC) which represent the specification text. Computer software is treated as Computer Software Configuration Items (CSCI's) with Computer Software Components (CSC's) and Units written in Source_Code (SC).

4. COMPATIBLE OPERATING ENVIRONMENTS

After satisfying the common problem and sufficient solution criteria, the issue of compatible environments remains. This has a scope which includes the processing type, processor type/scale, language, operating or run_time_system, requisite libraries, test results instrumentation system, and development support computer system.

Compatibility can occur with a range that varies from high_compatibility for a machine-independent software component that can be moved easily between systems, to low_compatibility where only the software component design can be moved manually between systems. Examples of varying compatibility are listed in Table 2. No language has a monopoly on highly compatible reuse – this implies reuse is language independent – although some languages are easier to reuse than others.

EWB-REE061589-2/6

5. CREATION OF AN ACCESSIBLE MARKET OF SOFTWARE COMPONENTS

Reuse is directly dependent on the availability of a library of software components at all levels. Widespread or "popular" reuse will only occur if there is a marketplace for CSC's/CSU's in SC and SPC's in DC. The reuse marketplace is sustainable if:

1. the customer procures components as well as systems
2. the contractors deliver components as well as systems
3. a forum is available in which to exchange components
4. the components are delivered in, and described in, a recognizable, standard form on which a "comparison shoppers" decision can be made

6. PROBLEM SIMILARITY VIA GENERIC SYSTEM MODELS

One method of enhancing reuseable component descriptions in a recognizable form is to create generic models for realtime processing systems of each generic type: radar, EW, EO, CNI, C³I, ESM/I, mission, weapon_control, telemetry, etc. These generic models would have to use standard realtime system models (TBD) and be comprehensive enough to cover all current and anticipated applicable generations and operating concepts of each system type. The capabilities in each system would be described by the SPCs, CSCs, and CSUs comprising the system. Hopefully, these components would be selected from the reuse marketplace. As anticipated new systems develop, their new components would be added to the reuse marketplace with the appropriate extensions to the generic models.

At the Westinghouse ESG, a Concurrent Managed Mode (CMM) model and a realtime pipeline (RTP) model is being used for radar system development. Generic extensions of the CMM and RTP to domain analysis and reuse evaluations are being considered.

7. REUSE IS ONE PART OF A SOFTWARE ASSEMBLY LINE

The System_Requirements, Timeline_Sizing, and Process_Composition expert systems of the Software Assembly Line (SAL) of Figure 1 could utilize components from the Reuse Marketplace. These three expert systems are only about 1/3 of the total effort required to automate the software development as envisioned in Figure 1. Significant inventions are required in several areas. Only a government/industry could attempt to create the capability of Figure 1. Current management practice would use different parts of the SAL at the Software Development Facility (SDF) and the Software Test Facility (STF).

TABLE 1 - Levels for System and Software Reuse

LEVEL	DOD-STD-2167A	MIL-STD-483
Proposal, System/System_Segment, or Configuration_Item (CI)	Proposal* Specification (SPC)	Proposal* A_Spec
System Capability or Mode	MDD*	MDD*
Requirements or Development	SRS/IRS (Functions)	B5_Spec (Functions)
Preliminary Design Detailed Design	SDD/IDD (SPC,CSC's) SDD/IDD (SPC,CSC's)	
Source code of Product	CSU's	C5_Spec (CPC's)
Test Plans/Procedures	STD (SPC)	Specification -
* non 2167A/483 document used by WEC System & Software Engineering		

TABLE 2 - Compatibility Variations in Reuse

High Compatibility

CSC in FORTRAN, JOVIAL, C, Ada – same processor type/scale/OS/library
SPC in ASCII Text

Medium Compatibility

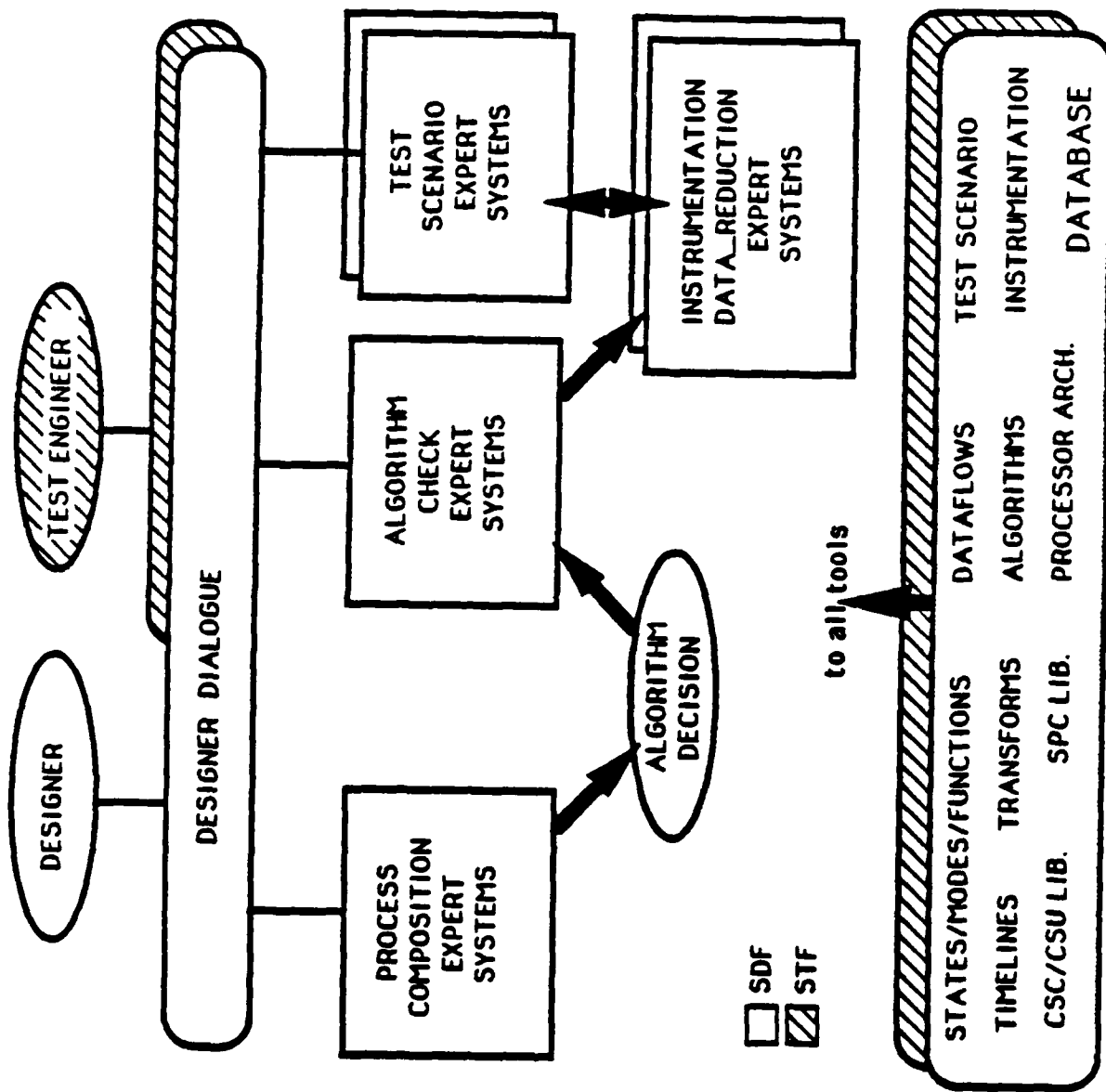
CSC in FORTRAN, JOVIAL, C, Ada requiring special O.S. or Library
or using a different processor type/scale

Low Compatibility

CSC in Assembly Language, RTL, or Microcode

"Design-Only" Compatibility

CSC in Assembly Language, RTL, or Microcode
with ISA unavailable or Development System not transportable
CSC in FORTRAN, JOVIAL, C, Ada to a system where the compiler,
Run_Time_System, and Development System are not available



SOFTWARE ASSEMBLY LINE (SAL) on SDF and STF

FIGURE 1

8. DOMAIN ANALYSIS EXAMPLE - AIRBORNE RADAR

Airborne fire control radar domain analysis includes the functional scope of the radar hardware and its technology level, the system functions/modes, future radar technology trends, the operating concepts, the radar system interfaces, and the role of a CSC/CSU in a radar operation.

The radar architecture is the functional scope of the radar hardware and its technology level (i.e. the generation of radar development) ; it is a key part of the radar domain. Eight generations are apparent:

- Gimbaled, analog non-coherent scanning; monopulse
- Gimbaled, analog coherent (PD) scanning; monopulse
- Gimbaled, digital coherent (PD) sequential lobing; monopulse
- Passive array, digital coherent (PD) monopulse
- Active array, digital coherent (PD) monopulse

The radar system functions or modes could include:

- Low, Medium, High PRF (Non-coherent, coherent)
- Search /Track
- Identification
- Kill Assessment
- Ground Map / DBS / SAR / ISAR
- Air-To_Ground Ranging (AGR)
- Terrain Avoidance /Follow (TA /TF)
- Missile Guidance
- Missile Warning
- Passive

Each function or mode has functional performance attributes within which a software item or component must fit. These attributes for a candidate CSC are to be examined by a software reuse browsing or recomposition tool. For a radar CSC, the attributes of a CSC's description can be listed within the scope of the radar problem domain and its elements:

- Purpose and function in the CMM/RPO and Operating Concept
Single mode/interleaved operation
vs.
Avionics Interface, Manage, Schedule, Control, Measurement,
Signal Processing (SP), Data Processing (DP) steps
- Measurement Structure
- Performance per antenna, receiver, and power gains
- Target capacity
- Timing/sizing with respect to the CMM/RPO
- Interfaces - Mission, INS, EW, radar equipment
- Related CSC's/Units and Specification Code (SPC)
- Applicable processor environments
- Applicable development environments

GTE

Software Reuse

for

Information Management Systems

Joel Cohen

14 April 1989

**GTE Government Systems Corporation
Strategic Electronic Defense Division
National Center Systems Directorate
1700 Research Boulevard
Rockville, Maryland 20850-3181**

GTE Government Systems, National Center Systems Directorate (NCSD) in Rockville, MD has gained experience in the area of software reuse as a result of an ongoing Imagery Information Management System (IIMS) Reuse IR&D project. This IR&D effort began in 4Q88 and grew out of an examination of existing information management systems supporting imagery. Many of the currently available systems consist of multiple, disjoint databases, whose information content must be manually interpreted and integrated by users. They require an excessive input of user/analyst time that should be spent on analysis and interpretation of imagery intelligence data.

In order to attack the IIMS problem, a more economic way of building such a complex system needed to be identified. Key to such a development is the ability to reuse existing information management system and software components. Thus, the IIMS IR&D project was initiated with the following objectives:

- analyze and produce a domain model for IIMS, generating a generic IIMS architecture and an application classification scheme into which reusable components can be categorized;
- document the feasibility, procedural differences, and steps to be taken to incorporate a component-based development into DOD lifecycle standards;
- investigate tools and techniques for identifying, retrieving, and incorporating components (reusable and commercial-off-the-shelf software) to develop IIMS;
- populate a software library with IIMS related components that can be used to prototype or develop future systems; and
- develop an IIMS application prototype as a proof of concept application and to provide first iteration feedback on the domain model and development methods developed in the first two tasks.

The approach to investigation under the project includes five major tasks:

1. IIMS Domain Analysis;
2. Component-based Development Methodology;
3. Software Reuse Tools and Techniques;
4. Establishing an IIMS Reuse Library; and
5. IIMS Prototype Development.

A major assumption going into the IR&D project was that an in-depth examination of each task would be sacrificed in order to gain as much experience as possible in each area. All of the tasks are currently in various stages of completion, as described below. Each task was initiated based on resource and time constraints rather than on a logical progression from one task to another.

IIMS DOMAIN ANALYSIS

The IIMS Domain Analysis task focused on the analysis of the IIMS application domain following a methodology developed by James Gish, Gerald Jones, and Ruben Prieto-Diaz at GTE Laboratories in Waltham, Massachusetts and documented in a technical report entitled "Domain Analysis: Procedural Model Refinement and Experiment Proposal" by Messrs. Gish and Prieto-Diaz [GISH88]. The purpose of the domain analysis task was to identify the objects that make up an IIMS domain, classify them, and

frame them in a structure that facilitates creation of system models of the IIMS domain. The suggested use of the models is to identify components and their relationships within an IIMS. Identified components become candidates for reuse when building similar systems. The analysis proceeds from the characteristics of a specific system to the development of general or generic models describing the generic components and their interaction.

The Domain Analysis methodology pursued deals with the identification of objects, functions, and relationships common across the domain. The domain objects, functions, and relationships are used to define a domain taxonomy and a domain model. The methodology involves eight steps, seven of which were performed under this task. The eighth step, involving the definition of a domain language, was not attempted due to time and budget constraints. The seven steps pursued include:

1. Select Specific Functions/Objects;
2. Abstract Functions/Objects;
3. Define Taxonomy;
4. Identify Common Features;
5. Identify Specific Relationships;
6. Abstract the Relationships; and
7. Derive a Functional Model.

The flow of these steps is shown in figure 1, from [GISH88].

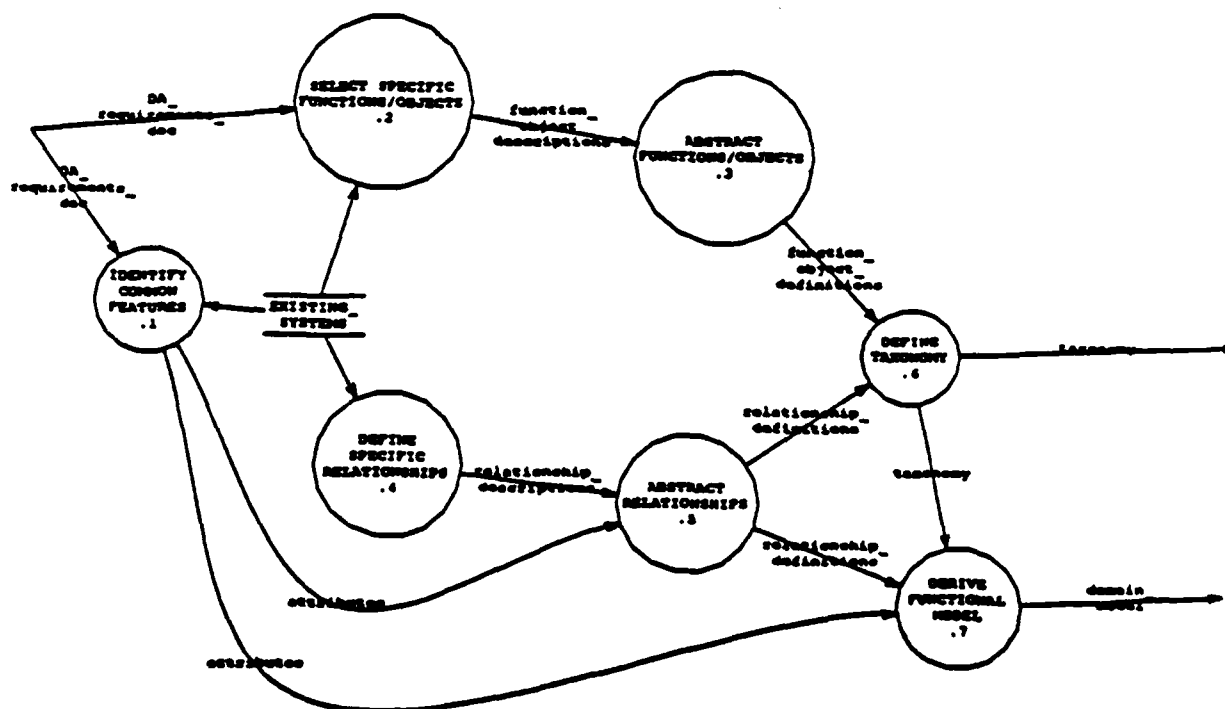


Figure 1. Domain Analysis Transforms Requirements into a Taxonomy and Models

Before executing any of the above steps, the scope of the domain of the system(s) to be analyzed was defined in order for the domain model to be both sufficiently general yet detailed enough to express applications of broad scope within the domain. This domain definition provided a basis of understanding before proceeding with the analysis steps.

After the domain was defined, the first step in domain analysis was to identify specific functions and objects within the domain. Domain requirements were analyzed and lists of objects within the domain and functions related to the objects were created. The objects and operations or functions were then grouped, based on an abstraction of their common attributes, and formed into classes. The identification and definition of these classes by grouping and classification constituted a taxonomy.

The first taxonomy derived was a hierarchical model. The taxonomy was then reorganized into facets which are perspectives or points of view of a particular class. The faceted taxonomy offers more flexibility than a single hierarchy and provides a more comprehensive definition of the domain. Both a taxonomy of functions and a taxonomy of objects were defined in this way.

After the preliminary taxonomies were generated, existing systems were examined to try to isolate their commonalities. A list of common features was then generated. Specific relationships between objects and functions were also extracted from existing systems. The relationships were then abstracted in order to determine how relationship descriptions can be generalized within the framework of common features. In the process, the specific relationships were mapped onto the common features. The listing of common features, specific relationships, abstract relationships, and the taxonomies were then used to derive a functional data flow model of the system.

As an added step to the methodology, an Entity-Relationship Model was developed to clarify the relationship between objects or entities within the system and as another way to clarify the functions of the IIMS domain.

IIMS DOMAIN ANALYSIS CONCLUSIONS

Several conclusions about domain analysis and the methodology used were noted at the completion of the Domain Analysis task. The expectations going into the domain modeling phase of this project were that the resulting model would provide a basis to develop an IIMS prototype and that the taxonomy resulting from the domain analysis could be used for developing the faceting scheme for a library of reusable software. The domain modeling effort was only partially successful in reaching these goals. The lack of success can be attributed to several factors, including inexperience with the domain analysis methodology, and the planned approach not to do an in-depth domain analysis, but rather to push through the methodology at a more superficial level in order to gauge its overall effectiveness.

The methodology proved to be effective in gaining an understanding of the domain of interest, namely imagery information management systems, and resulted in initial models for developing the prototype.

However, the functional model proved not to be the most effective or comprehensive model for the prototype development task.

The functional model documented in the domain analysis phase proved to be sufficiently generic to other types of information management systems. Having the imagery functionality in the models provided the interfaces and place holders where other, nonimagery applications could be substituted for the imagery. Although information management may be the broader (more generic) domain, narrowing the focus to data specific (imagery, less generic) applications helped define domain boundary points, and resulted in a model that is useful (more generic!) to other types of information management systems.

The other major unexpected conclusion resulting from the IIMS domain analysis is that multiple taxonomies or classification schemes are necessary to support reuse at different levels of abstraction; from the design of system architectures, through the construction of the system reusing specific software components. The domain analysis methodology was effective in defining the system domain but did not result in a taxonomy that was useful for classifying components to build the system. The methodology pursued stopped short of defining guidelines for transforming the resulting generic system taxonomy into a useful scheme for classification of specific pieces of software.

The domain analysis taxonomy appears to be most useful for the earlier development phases where system requirements are analyzed for commonality with parts of the domain. The domain analysis models and taxonomy are useful for determining where or if there is commonality between systems. However, trying to build the classification scheme for the library of software components from the domain analysis taxonomy would not work. The facets for classifying and using design level specifications are sufficiently different from the facets used for software components to require different library taxonomies. This was not obvious from the domain analysis methodology or from the guidelines on building reusable libraries until these methods were attempted in practice.

The overall domain analysis methodology worked well to describe common elements of imagery information management systems and to point towards areas to investigate for reuse during actual design and implementation. Two additional techniques are necessary: an approach for building phase-specific (system architecture, system design, subsystem implementation) reuse libraries based on the domain analysis; and a way of linking system and high-level design components classified during domain analysis to the actual software components that can be reused to construct systems within a domain.

COMPONENT-BASED DEVELOPMENT METHODOLOGY

The basic premise of the component-based development methodology is that each unique system development is really an instantiation of functionality from one or more application domains. (An application domain is a class or family of systems that share common characteristics.) This commonality creates the opportunity to identify reusable components.

This task is intended to define in procedural terms the impact of the component-based development methodology on the standard DOD development methodology requirements. The current status of the task is that an initial look at the overall development methodology and how reuse refines the methodology has been started.

SOFTWARE REUSE TOOLS AND TECHNIQUES

The approach to this task builds on previous work that developed tools for software component cataloging and retrieval. Beginning with the utilization of the Asset Librarian System (ALS) developed by GTE Laboratories, this task will extend and complete software librarian capabilities currently available. The ALS tool is in the process of being rehosted from its current PC-based environment using Oracle and a proprietary windowing product to a SUN UNIX environment using SQL and the X Window System. A goal of this task is to produce a portable librarian toolkit with which customized reuse applications can be developed.

ESTABLISHING AN IIMS REUSE LIBRARY

This task and the development of a prototype IIMS were intended to work together as an iterative process. Under this task, software components have been identified and described in preparation for use of the ALS. As a result of the conclusion reached in the Domain Analysis task that there is a need for a separate classification scheme derived from the components as opposed to one derived from the domain, this task began with the identified software components and guidelines on building an ALS library [PRIET89]. An initial classification scheme has been derived and the library is ready to be populated with components.

IIMS PROTOTYPE DEVELOPMENT

The intent of this task was to apply the component-based development methodology to a small scale software development effort, using the components in the ALS library. The prototype designed represents the man-machine interface for a typical imagery analyst whose job is to do research using textual reports, images, and maps to produce some type of imagery report. The prototype is being developed in a SUN environment using SQL and the X Window System. This task wound up preceding the methodology and reuse library tasks. As a result, lessons learned in developing the prototype for and with reuse will be captured and used as input to these two tasks.

CONCLUSIONS

GTE NCSD has spent a good deal of effort addressing the problems of reuse in practice. We are applying reuse to a practical problem and are committed to continuing the investigation of reuse in the future. We are interested in sharing our experiences with other users in order to share our concerns and lessons learned, and to acquire other users' practical lessons learned. We are very interested in attending the Reuse In Practice Workshop and look forward to participating with other interested users.

REFERENCES

- [GISH88] James W. Gish and Ruben Prieto-Diaz. "Domain Analysis: Procedural Model Refinement and Experiment Proposal". Technical Note No.:87-126.05, GTE Laboratories Inc., 40 Sylvan Rd., Waltham MA 02254, April 1988.
- [PRIET89] Ruben Prieto. "Building a Library for Reusable Software". Technical Report TR-016-12-88-126, GTE Laboratories Inc., 40 Sylvan Rd., Waltham MA 02254, December 1988.

The Role of SADT

in Domain Analysis for Software Reuse

Position Paper

Ernesto Guerrieri

Theodore B. Ruegsegger

SofTech Inc., Waltham, Massachusetts 02154-1960

4 May, 1989

Recently, an internal group¹ at SofTech was reviewing Prieto-Díaz's paper on Domain Analysis [PRIETO-DÍAZ87b]. It became evident that there was a similarity between the proposed domain analysis process and the SADT² modeling process [ROSS85, MARCA86]. In the past, SofTech had used SADT to perform a "domain analysis" [RUEGSEGGER87]. In this paper, we would like to show the similarity that exists between the two processes and that SADT can play a role in performing domain analysis.

Some of the key points that Prieto-Díaz states in [PRIETO-DÍAZ87b] are:

- There is "no methodology or any kind of formalization" for domain analysis. The article provides data flow diagrams for a recommended process.
- Potentially reusable items are difficult to understand and to adapt.
- The use of domain analysis to "capture the essential functionality" will make the items more likely to be reused.

¹The Software Reusability Study Group meets on a regular basis to review and discuss topics on software reuse.

²Structured Analysis Design Technique.

- The domain analysis process is similar to knowledge acquisition, modeling, and object oriented programming.
- Domain analysis occurs prior to the Systems Analysis phase. It takes a more general or abstract viewpoint.
- The domain analyst searches for common characteristics (i.e., objects and operations).
- A "domain specific language" is recommended (with special syntax and semantics). Classification helps to develop one.
- A basic problem in domain analysis is defining a domain's boundary.

SoftTech performed for ISEC³ an assessment of the potentials for software reuse in traditional MIS applications. One of the findings from this study [RUEGSEGGGER87] was that:

"SADT is a useful method for the development, evaluation, presentation, and documentation of generic functional architectures. The "viewpoint" principle of SADT is an aid in making analogies among purportedly dissimilar systems."

Neighbors defines a domain as "the encapsulation of a problem area" and the domain analyst as "the person who examines the needs and requirements of a collection of systems which seem *similar*" [NEIGHBORS84]. As a consequence, we looked at SADT from a domain analysis perspective [FELDMANN89] and concluded that:

1. The key intellectual challenge in both processes is in recognizing and making commonality practical.
2. SADT is usable for domain analysis, but it lacks the domain specific language. It was noted that a domain specific language can be generated for an SADT model via FEO⁴ diagrams.

³U.S. Army. Information System Engineering Command.

⁴FEO (For Exposition Only) diagrams are diagrams that contain anything needed by an author to illustrate a point associated with an SADT diagram.

3. The common module identification in SADT is the same search for commonality as is performed in domain analysis.
4. SADT's syntax for common modules are the "down (call) arrows" and the "interface FEOs."
5. SADT is not limited to one domain. It can model interfaces between domains.

The SADT modeling process places a heavy emphasis on interviewing (i.e., information gathering), bounding the subject, determining the purpose and viewpoint of the model, and generating the data and activity lists. These steps recognize the need for a rigorous procedure due to the intellectual challenge in recognizing and forming commonalities.

A domain specific language is needed to communicate between the analyst and the domain expert. SADT's generic notation does not give that "look and feel" impression to domain experts reviewing a model of their domain. This can be achieved in SADT via a glossary of terminology and the FEO diagrams.

Experience has shown that the decomposition of one of the boxes in an SADT model should stop when the box is very similar to another box in the same model [MARCA88]. Two boxes are similar if they perform roughly the same function and have almost the same number and types of inputs, controls, and outputs. This search for commonality is similar to the identification of common features in domain analysis.

The appropriateness of SADT for domain analysis was realized by Douglas Ross several years ago [ROSS85] when he stated:

"SADT is an extremely powerful methodology for working out a clear-cut understanding of an at-first obscure and nebulous complex subject, documenting that understanding, and then communicating that understanding to others. ... SADT can provide the framework for a problem-solving methodology for any kind of problem."

References

- FELDMANN89** SADT presentation at the Software Reusability Study Group by Clare Feldmann, SofTech, Inc., Waltham, MA, April 18, 1989.
- FREEMAN87** "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," Freeman, P., **IEEE Transactions on Software Engineering**, Vol.SE-13, No.7, pp.830-844, July 1987.
- MARCA88** "SADT Structured Analysis and Design Technique", Marca, D.A., and McGowan, C.L., McGraw-Hill Book Company, New York, New York, 1988
- NEIGHBORS84** "The Draco Approach to Constructing Software from Reusable Components," Neighbors, J.M., **IEEE Transactions on Software Engineering**, Vol.SE-10, No.5, pp.564-574, September 1984.
- PRIETO-DÍAZ87a** "Classifying Software for Reusability," Prieto-Diaz, R., and P. Freeman, **IEEE Software**, Vol.4, No.1, pp.6-16, January 1987.
- PRIETO-DÍAZ87b** "Domain Analysis for Reusability," Prieto-Diaz, R., **Proceedings of COMPSAC 87**, Tokyo, Japan, October 1987, pp.nn-nn.
- ROSS85** "Applications and extensions of SADT," **IEEE Computer Magazine**, Vol.15, No.4, pp.25-34, April 1985.
- RUEGSEGGER87** "RAPID: Reusable Ada Packages for Information System Development," Ruegsegger, T., **Technology Strategies '87 Proceedings**, January, 1987

Features Analysis: An Approach to Domain Analysis¹

Kyo C. Kang

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
NET: kck@sei.cmu.edu

Abstract

A domain analysis was performed at the Software Engineering Institute as part of a reuse experiment. The analysis was called features analysis because of its heavy emphasis on the analysis of functional features. The goal of the analysis was to identify and represent a generalized functional model from which software requirements can be derived and based on which reusability of components can be evaluated and classification of components can be made. Some of the experiences from the analysis are: (1) the domain analysis provided opportunities for experts to consolidate and organize their domain knowledge and for non-experts to learn about the domain, (2) analyzing the functional features was an effective way to determine the product commonality and the scope of the domain analysis, and (3) there is no adequate mechanism for representing a domain model to support reuse through the requirements analysis phase.

The purpose of the domain analysis was to investigate the concept and feasibility, and there was no "formal" approach that was followed. A conceptual modeling method which is based on the analysis of the "universe of discourse" is proposed in this paper as a domain analysis method.

1. Introduction

The Application of Reusable Software Components (ARSC) project at the Software Engineering Institute (SEI) performed a features analysis as part of a reuse experiment [ARSC89]. The features analysis [PERRY88] is a type of domain analysis, in which features (e.g., functions, objects) of similar systems in the same domain are analyzed. Because of its heavy emphasis on the analysis of features, the domain analysis was called a features analysis in this project. The goal of the features analysis was to identify and represent a generalized functional feature model

of a family of systems with parameterization to accommodate the differences. The analysis was performed to provide a basis for the requirements analysis, evaluation of the reusability of components, and classification of reusable resources.

The features analysis was performed by two domain experts and two non-experts based on their domain knowledge and using the documents from the Common Ada Missile Packages (CAMP) project [MCDON85a] and the Program Performance Specification (PPS) [MCDON85b] of

¹The Software Engineering Institute is a federally funded research and development center sponsored by the Department of Defense under contract to Carnegie Mellon University. This paper is approved for public release.

the target system. The domain experts took the top-down approach and defined a high-level functional feature model based mostly on their domain knowledge. The non-experts took the bottom-up approach depending heavily on the CAMP documents and consultation with the experts. They identified the functional features implemented by the CAMP components and consulted with the experts to verify that the features covered general problems.

The analysis results were represented using the activity charts of Statemate [ILOGI87], which was adopted as the requirements analysis tool in this project. A sample activity chart is included in Exhibit 1.

Some of the lessons learned from the analysis are discussed in section 2. Our perspective on domain analysis is described in section 3, followed, in section 4, by an outline of a methodology proposed for domain analysis.

2. Lessons Learned

The features analysis was performed to investigate the concept and the feasibility, and it was never intended to generate a complete domain model. However, we learned a few lessons which are summarized in this section.

One of the problems encountered at the beginning of the analysis was to determine the scope of the domain. One domain expert was knowledgeable in cruise missiles whereas the other expert was knowledgeable in air-to-air missiles, and they often used the same terminology for different meanings. As they resolved the differences, it was realized that, although both types of systems share some common features at the high level, there were not enough commonalities that could be adequately represented by one functional features model. Guidance and navigation methods, which are major features of missile systems, were different and they had to modify their definitions to

reconcile and come up with a single model. We believe that determining the scope of the domain is a necessary first step of the analysis, and that analyzing the features of products is an effective way to determine the scope.

Experts with different backgrounds often used different terms for the same thing or the same terms with different meaning. For example, the term "navigation" had a different meaning to the air-to-air missile expert than to the cruise missile expert. To avoid possible misinterpretation of a domain model, we believe that a dictionary which includes the definitions of the keywords used in the model and description of the model should be produced during the domain analysis.

As stated previously, we used the Statemate activity charts as the domain model representation mechanism. The rationale behind the choice is that we already chose to use Statemate for the requirements analysis and we wanted to derive the requirements specification from the features model by selecting the features that are appropriate for the target system. Although, we did not have any problem describing the features and their static structure, it was very difficult to represent logical relationships between the features using Statemate. We wanted to specify the constraints and composition rules among the features (e.g., the feature A must be, or must not be, selected when the feature B is selected). Also, we wanted to specify if a feature is optional or mandatory. (We understand that Statemate was not designed to represent the kind of information we wanted to express.) We believe that this is a general shortcoming of most of the CASE tools available today to support the domain analysis. In order to have reuse occur at the requirements level, we need a mechanism that can adequately represent the general functionality of software systems from which specifications of a specific software system can be generated.

Although, we could not produce the requirements specification directly from the

domain model, the knowledge gained during the features analysis greatly helped us to do the requirements analysis. The developers could acquire enough domain and reusable components knowledge to define the requirements. Based on our experience we believe that the domain analysis or reexamination of the domain model should be a standard phase in the software life-cycle.

Our perspective on domain analysis is described in the following section.

3. Perspective on Domain Analysis

Our perspective on domain analysis is described in this section by answering two questions: what is domain analysis? and why do we need to do it?

What is domain analysis?

Domain analysis is a phase in the software life-cycle where a *domain model*, which describes the common functions, data and relationships of a family of systems in the domain, a *dictionary*, which defines the terminologies used in the domain, and a *software architecture*, which describes the packaging, control, and interfaces, are produced. The information necessary to produce a domain model, a dictionary, and an architecture is gathered, organized, and represented during the domain analysis.

Domain analysis is related to requirements analysis but it is performed in a much broader scope and generates different results. It encompasses a family of systems in a domain, produces a domain model with parameterization to accommodate the differences, and defines a standard architecture based on which software components can be developed and integrated. A domain model and an associated dictionary represent the domain knowledge, and an architecture represents the framework for developing reusable components and for synthesizing systems from the reusable

components. An ideal domain model and architecture would be applicable throughout the life-cycle from requirements analysis through maintenance.

Why do we need to do domain analysis?

As the areas to which computers are applied become larger, one of the problems faced by the industry is that it is often difficult to find software engineers who have the required application domain knowledge. Reuse of application domain knowledge is becoming an important issue in software engineering. The purpose of domain analysis is to gather and represent application domain knowledge in a model and to develop an architecture that shows how the problems in a domain are addressed in software systems. A domain model unifies and consolidates the domain knowledge which may be reused in subsequent developments.

More and more organizations consider software as an asset that can provide an important edge in business competition. Therefore, identifying areas that will maximize the return on software investment is an activity that encompasses both business planning and software engineering. The business planning activity identifies future products, and the domain analysis activity identifies the product commonality and potential software assets. The information on the software assets can be fed back to future business planning. Also, the product commonality information enables large-grain reuse across the products.

The productivity and quality improvement from reusing components built for the purpose of reuse is much greater than that from components developed without reuse in mind. However, in order to build reusable components, the contexts in which the reusable components will be used must be understood and the reusable components must be designed to accommodate the contextual differences. A domain model and an architecture

define the contexts for developing reusable components.

In summary, the output from the domain analysis can be used to:

- define the context in which reusable components can be designed and developed,
- ascertain reusability of candidate components,
- identify and develop software assets,
- provide a model for managing (classifying, storing, and retrieving) the software assets, and
- provide a framework for tooling and systems synthesis.

As noted previously, the purpose of the features analysis in this project was to investigate the concept and feasibility, and there was no formal approach followed by the project. A repeatable method is needed to evaluate and improve the domain analysis process. In the following section, a conceptual modeling technique is proposed as a domain analysis method.

4. An Approach

A number of systems modeling techniques have been developed and used in the database area [BRODI84]. Of those techniques, a conceptual modeling technique is summarized from [NIJSS76] and [SCHIE79] and proposed as a domain analysis method in this section.

Construction of a system starts by perceiving a conceptual model which may be derived from existing "reality" [NIJSS76] or from a hypothetical system. *Reality* is defined to be a system (or systems) in existence and consists of entities and relationships between the entities [CHEN76]. An entity is either a physical entity or a concept. Through a human perceptor, a real system is perceived and then described by "naming": entities, properties, and relationships which are perceived for a system are named. An entity is

characterized by its properties, and some of these properties (for example, name) may be used as identifiers. The end result of naming is called a "perceived reality."

The perceived reality may contain names which are not of concern for the target system. Those unnecessary names are eliminated and what is left is the "universe of discourse." The universe of discourse is an objective system for which the description will be made, and within which dialogue between the system developers will be limited.

The entities in the universe of discourse are classified into homogeneous sets (classes or types) of entities; homogeneous in the sense that all entities in the same class have some properties in common. Each entity classified as such is named as an *entity type*, which is a unit in conceptual model construction. Relationships existing between the entities are also classified into relationship types, and these types are defined between the entity types. Properties of the entities in a class are classified into property types, and for each property type, possible values are defined. A conceptual model contains the description of the abstracted universe of discourse. The process is summarized in Figure 1 below.

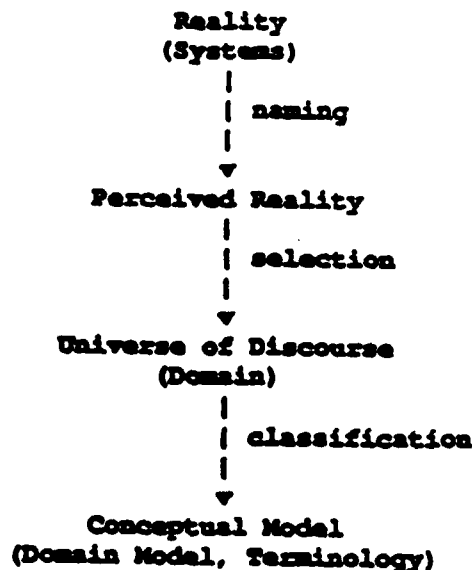


Figure 1

A conceptual model consists of abstract elements of a system (or systems), such as entity, attribute, and relationship types, and rules and constraints between the elements. The entity-relationship model [CHEN76] or semantic data modeling techniques [MCLEO78] may be used as a conceptual model schema language. It is demonstrated in [KANG82] that an extended entity-relationship model can be used as a meta language to define systems specification languages.

5. Summary

Domain analysis is an activity to produce a domain model, a dictionary of terminologies used in a domain, and a software architecture for a family of systems. These outputs from the domain analysis:

- facilitate reuse of domain knowledge in systems development,
- define the context in which reusable components can be developed and the reusability of candidate components can be ascertained,
- provide a model for classifying, storing, and retrieving software components,
- provide a framework for tooling and systems synthesis from the reusable components,
- allow large-grain reuse across products, and
- can be used to identify software assets.

Based on our experience with a domain analysis (called features analysis in this project) and the potential benefits from it, we believe that domain analysis should be a standard activity in the software development life-cycle.

References

[ARSC89] ARSC, *An Experiment to Analyze a Reuse-Based Software Development: Detailed*

Design, Software Engineering Institute (technical report in preparation), April, 1989.

[BROD184] Brodie, M.L., Mylopoulos, J., Schmidt, J.W., editors, *On Conceptual Modelling*, Springer-Verlag New York Inc., 1984.

[CHEN76] Chen, P.P., "The Entity-Relationship Model - Toward a Unified View of Data", *ACM Transactions on Database Systems*, vol. 1., no. 1, p.9-36, March 1976.

[ILOG187] i-Logix Inc., *STATEMATE: The Language of Statemate*, i-Logix Inc., Burlington, MA., March 1987.

[KANG82] Kang, K.C., *An Approach for Supporting System Development Methodologies for Developing a Complete and Consistent System Specification*, Ph.D. Dissertation, The University of Michigan, Ann Arbor, MI., 1982.

[MCDON85a] McDonnell Douglas Astronautics Co., *Common Ada Missile Packages (CAMP): Overview and Commonality Study Results*, McDonnell Douglas Astronautics Co., 1985.

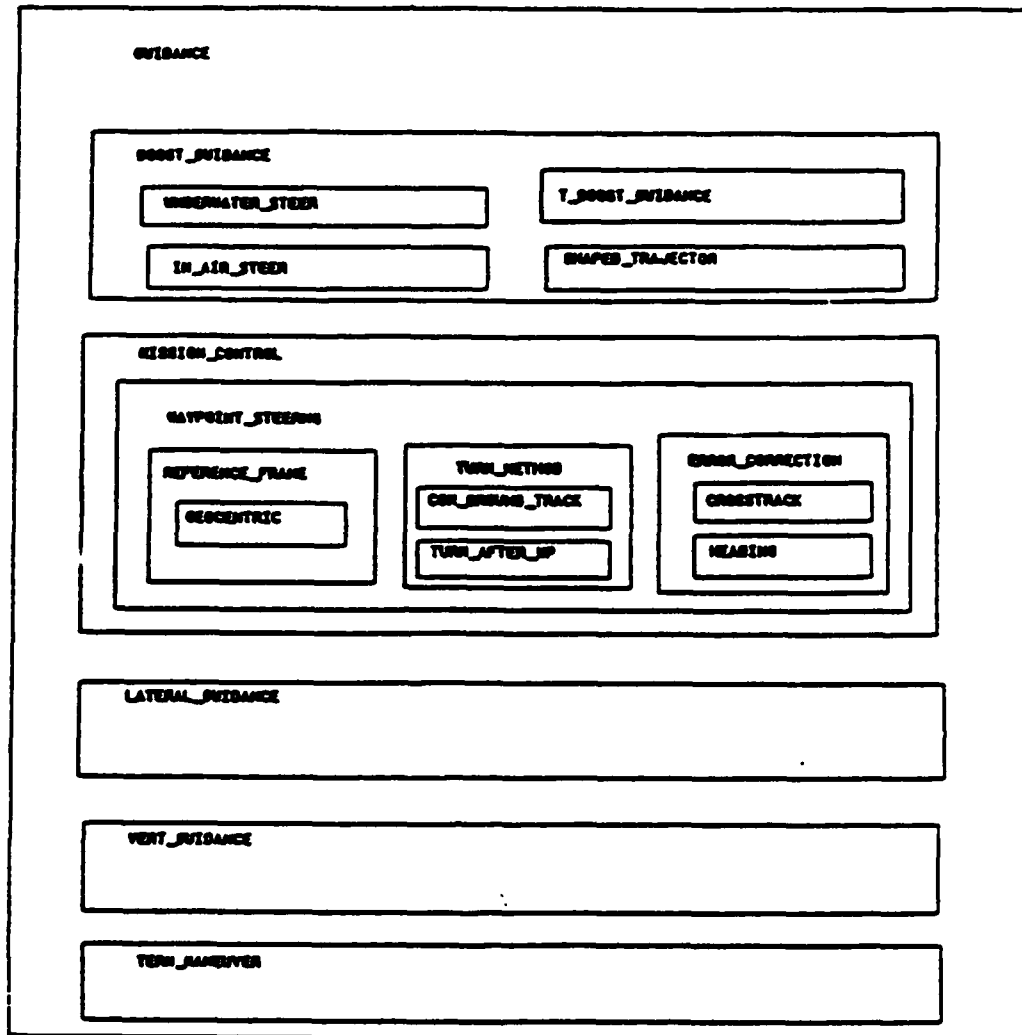
[MCDON85b] McDonnell Douglas Astronautics Co., *Computer Program Performance Specification Cruise Missile Land Attack Guidance System BGM-109C*, McDonnell Douglas Astronautics Co., 1985.

[MCLEO78] McLeod, D., *A Semantic Data Base Model and Its Associated Structured User Interface*, Ph.D. Dissertation, MIT, Cambridge, MA., 1978.

[NIJSS76] Nijssen, G.M., "A Gross Architecture for the Next Generation Database Management Systems", *Modelling in Database Management Systems*, North-Holland Pub. Co., p.1-24, 1976.

[PERRY88] Perry, J., *Perspective on Software Reuse*, Software Engineering Institute (CMU/SEI-TR-88-022), November, 1988.

[SCHIE79] Schiemann, A., "A New Approach to the Entity-Relationship Model", Proceedings of the International Conference on Entity-Relationship Approach to Systems Analysis and Design, p.383-408, December 10-12, 1979.



A Sample Functional Feature Model

Exhibit 1.

Application of Domain-Specific Software Architectures to Aircraft Flight Simulators and Training Devices

Kenneth J. Lee

Software Engineering Institute

Michael Rissman

Software Engineering Institute

1. Introduction

Work on domain-specific software architectures (DSSA) has been on-going at the Software Engineering Institute (SEI) in Pittsburgh, Pennsylvania, since 1986.¹ The primary goal of this project is to encourage the creation and use of canonical design specifications for typical, recurring problems in an application domain. A canonical design specification contains templates for software that ratifies a specific instance of a recurring problem. We are interested in canonical design specifications that embody principles of object-orientation. The project will be successful if it does nothing more than directly and indirectly generate examples of domain-specific specifications to object-oriented designs.

Aircraft flight simulators and training devices have been the primary application domain for our work. This paper describes a specification for a typical problem in the domain and discusses the benefits of both the particular specification and the use of canonical specifications in general.

1.1 Background

The effort in the domain began with our participation in the Ada Simulator Validation Program (ASVP), a research and development effort by two aerospace contractors to redesign and reimplement subsets of two existing flight simulators in Ada. The SEI project team supported the Air Force System Program Office² by attending reviews and acting as technical consultants. As part of our involvement, we developed a software architecture for flight simulator systems³. The software architecture has received acceptance in the domain and is being used by contractors involved in full-scale development efforts. Section 3 describes the results of this work in more detail.

¹ This work is sponsored by the U.S. Department of Defense.

² SPO for Training Systems (ASD/YW) at Wright-Patterson Air Force Base.

³ This work is described in detail in the SEI Technical Reports, *An OOD Paradigm for Flight Simulators*, 2nd Edition [2] and *An OOD Solution Example: A Flight Simulator Electrical System* [1].

1.2 Reader's Guide

This paper is too brief to be able to cover all aspects of our work in the domain. The SEI technical report, *An OOD Paradigm for Flight Simulators*, 2nd Edition, describes our work with a DSSA in the flight simulator domain; see [2]. The SEI technical report, *An OOD Solution Example: A Flight Simulator Electrical System*, describes the implementation of a typical simulator system using the OOD paradigm; see [1].

The concept of a DSSA is introduced in Section 2, which also discusses the approach used to develop the specification and some benefits of the approach.

Section 3 presents a view of the flight simulator domain and describes a typical problem and its specification. The section concludes with a discussion of some advantages of using DSSA.

Section 4 briefly describes the future direction the DSSA project will take.

2. Domain-Specific Software Architectures: Definition of Terms

This approach to building large software systems involves breaking the problem up into many smaller, recurring problems. Recurring problems are problems that occur more than once. One identifying characteristic of a recurring problem is that the problem occurs in several places within a system, often on different processors. In the flight simulator domain, a recurring problem is the description of aircraft systems in terms of objects, connections, and control mechanisms. The systems may be part of one executive running on a single processor or several executives on several processors. Another aspect of recurring problems is that they tend to be those parts of the system that act on many different instances of the same kind of data. For example, a simulator electrical system may have several hundred circuit breakers. A specification to a

recurring problem addresses all aspects of the problem from all occurrences of the problem in the system. Implementations of a canonical specification to a recurring problem are then similar in structure, behavior, and functionality.

A domain-specific software architecture is a set of specifications to recurring problems that characterize a domain. The specifications are represented diagrammatically. The diagrams are constructed using icons that represent domain-specific functional idioms and forms.

The specialized functional idioms represent classes of things from which design specifications can be formed. The things can be entities (for example, objects), connections between entities, or abstractions for grouping and updating related entities (for example, systems and executives). A functional idiom is the basis of functionality in the diagram. Functional idioms abstract the functionality of the design (see Figure 1). Rules of composition are associated with each idiom. For example, the rules for a connection are:

- ◆ A connection touches two objects, one at the head of the connection and the other at the tail of the connection.
- ◆ The label on a connection can list the elements of a composite.
- ◆ Connections may pass through a system symbol, but originate and terminate only at objects.

Forms are code templates satisfying the functionality of a functional idiom. The forms abstract the functionality of the implementation. A file containing a form will contain an Ada package specification, a package body, and a test procedure. The file contains placeholders for the name of the form, the Ada types used in the form, and so on. The placeholders must be globally replaced with appropriate values using an editor. Global replacement of the placeholders affects the specification, the body, and the test procedure. The resulting file is a compilable unit that can be tested (after compilation and linking) with the asso-

ciated test procedure. Also, forms allow for automation. Given a diagram and a diagram parser, automatic generation of software code would be feasible using the forms.⁴

When a design specification is created with icons representing functional idioms and forms are assigned to each functional idiom, we say that the design specification is a metaphor for the software system. Once a designer, an implementor, or a maintainer has learned the functional idioms and the forms, the design specification provides a detailed description of the system. In other engineering disciplines such a diagram, for example, a blueprint in architecture, is accepted as the design of the entity it represents. Practitioners can use the diagram and understand where a wall is to go, how high it will be, and how it will be constructed. We assert that this kind of diagram serves the same role for software systems.

The development approach using DSSA involves the following steps:

- ◆ Identify the recurring problem(s), e.g., aircraft simulator systems.
- ◆ Solve each recurring problem for a single instance of the problem, e.g., for an engine system.
- ◆ Review and validate the design using the design specification.
- ◆ Generalize each specification to produce forms.
- ◆ Verify the forms with respect to the design specification.
- ◆ Generate the rest of the instances of the specifications using the forms.

This is a depth-first approach. The approach provides several benefits:

- ◆ Design decisions are made once and applied to all instances of the specification.
- ◆ Multiple instances of the design, which would be subject to maintenance and enhancement, are not created before the design is validated.

⁴ We are not aware of a tool that could parse these diagrams. We do not have the resources to build such a tool. We would be willing to collaborate with tool builders interested in such a venture.

icon	functional idiom	functionality
rectangle	object	maps inputs to outputs ...
arrow	connection	moves effects between objects ...
round-cornered rectangle	system	groups related objects, provides update abstraction for the set of objects
gray-filled area	executive	groups systems; provides ordered update for the systems

Figure 1: Classes of Functional Idioms

- ◆ Documentation can be produced in a similar manner, i.e., instances of documentation can be produced for the initial instances and validated. Then as new specification instances are created from the forms, new instances of the documentation can be created from documentation forms.
- ◆ The amount of documentation produced will be less. The generalized specification can be described once. Each instance only needs to describe the qualities that make it unique. Pointers back to the description of the general specification suffice for all other qualities.

3. A DSSA in the Flight Simulator Domain

Our involvement with flight simulators began with the ASVP. The concepts embodied in the DSSA, developed during the program by the SEI team, were accepted by the contractors and endorsed by the Program Office. Subsequent work with contractors doing full-scale developments of simulators has matured the DSSA.

3.1 Domain View

A natural analogy exists between a flight simulator and the real world. A real-world aircraft is built of systems, and the systems are built from parts.⁵ These parts can be created explicitly in software. Thus, there can be a direct correspondence between the real-world entities and software entities. A specification that takes advantage of an objective view of a sys-

tem must also address those entities that are needed to run the simulator on a computer, e.g., connections for moving data, and executives for managing time.

The identification of an aircraft system as a recurring problem falls directly out of a cursory view of the domain. There are several systems in a simulator, e.g., an electrical system, a fuel system, a hydraulic system, an engine system. Each system is periodically updated. At the time of the update, the system must have access to the state of the world outside the system. When the system update is complete, the world must have access to the state of the system. Thus, a general specification for aircraft systems that addresses objects, connections, and update triggers should be applicable across simulator systems. The next section describes a specification to this recurring problem. See [1] and [2] for implementations of the specification.⁶

3.2 A Domain-Specific Software Architecture for Flight Simulators

This section describes an instance of a specification to a recurring problem for flight simulators. The recurring problem is the description of aircraft systems in terms of objects, connections, and control mechanisms. The specification defines a DSSA for this domain.

This specification is for an aircraft simulator engine system. But this is an instance of a specification to a recurring problem; thus, the characteristics of the

⁵ For example, an engine system is made up of burners, fans, turbines, and so on.

⁶ Other considerations that must be kept in mind include:

- ◆ real-time performance
- ◆ impact of compilation dependencies
- ◆ post-deployment software support

⁷ Systems outside the flight executive, e.g., the instrumentation system, are assumed to be part of other executives perhaps running on other processors. See the Paradigm report [2] for more information.

specification, including the diagram and the discussion, apply equally to the other systems in a simulator.

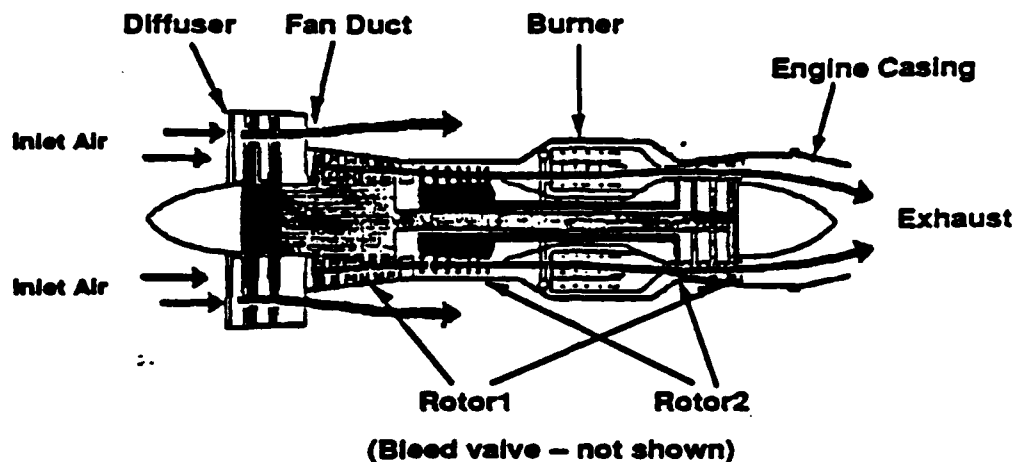


Figure 2: Turbofan Engine Cross-Section

The objects in the engine system are the diffuser, rotor 1, rotor 2, burner, fan duct, exhaust, engine casing, and bleed valve. System-level connections are shown, e.g., a connection moves discharge air pressure, discharge air temperature, and discharge air flow from the diffuser to the engine casing. Executive-level connections between objects in the engine system and objects in other systems are also shown, e.g., a connection moves a fuel flow value from an object in the fuel system to the burner object in the engine system.⁷

Figure 2 shows a turbofan engine in cross-section. The parts of the engine are labeled and the flow of air through the engine is shown. Figure 3 shows an engine system design specification that corresponds to Figure 2. Figure 3 is built using the icons discussed in Section 2.

The icons are put together according to the rules of composition associated with each idiom (see [2] for more information). There is a name correspondence between the labeled parts in Figure 2 and the labeled objects in Figure 3. There is also a functional correspondence. Air flows into the diffuser from the environment and passes through the engine casing. The engine casing allows the air to flow through the engine, interacting with each object in turn. Some en-

ergy is removed from the air by the fan duct and the turbine blades of the rotors. Energy is added to the air by the rotor fan blades and the burner (combustion chamber). Thrust, which drives the airframe, comes out through the fan duct and through the engine casing as exhaust. The name correspondence provides a basis for traceability.

The engine system design specification, Figure 3, depicts one system on the aircraft. Each system will have a similar specification that shows its connections to the outside world and the connections between its objects. The separation of connections from objects allows systems and objects to be independent. Independence permits separate development of systems and objects, defines natural divisions for assigning systems to processors, localizes details to allow for easier modifications, and allows replacement of systems and objects in toto.

The executive encompasses systems and manages time for the systems. Time is managed with a cyclic executive. The use of a cyclic executive, traditional in flight simulators, is a viable mechanism for simulating concurrency. An implementation, using this DSSA, allows for localization of scheduling information, which may ease load balancing (see [2] for further discussion).

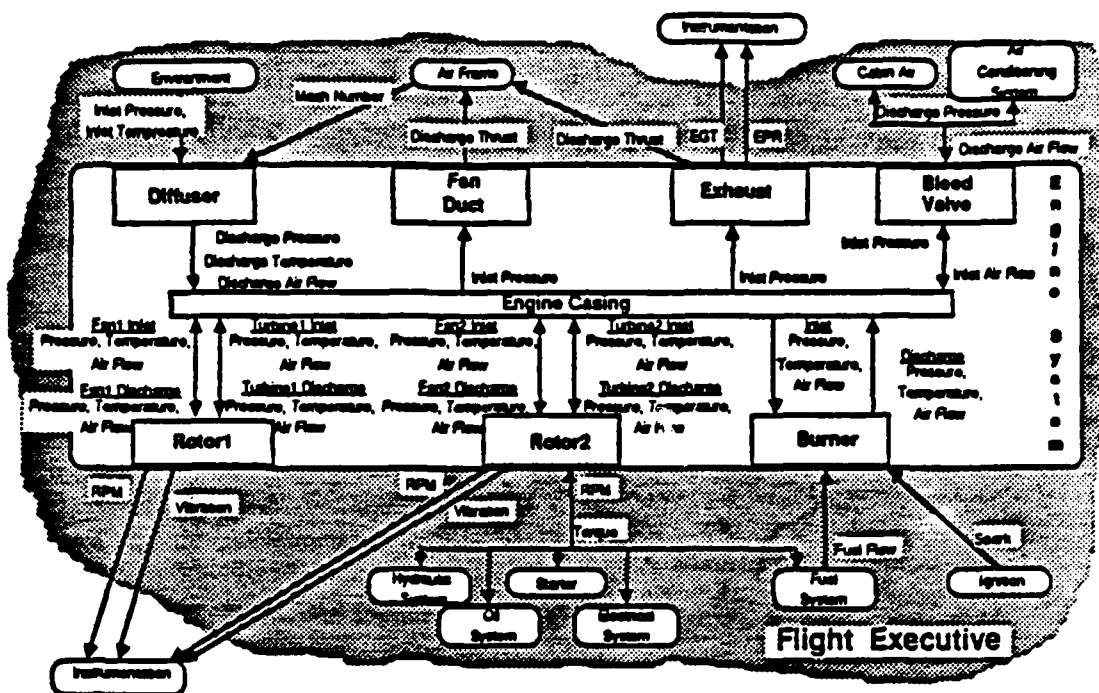


Figure 3: Turbofan Engine Design Specification

3.3 Advantages of a DSSA for Flight Simulators

Our expectations that DSSA and design examples would transition well and be reusable are becoming reality. The concepts have been accepted by the flight simulator community. Program Offices believe that requiring a DSSA early in the acquisition process is a reasonable and necessary factor in assuring the success of future programs. Contractors in the domain that were unaware of the benefits of DSSA, having created and used a DSSA for several months, now espouse the use of DSSA. The evolution of the basic ideas by the contractors will aid in maturing the technology.

Some comments from contractors working on full-scale developments:²

- ◆ A DSSA provides a common language. Engineers are able to discuss the design in terms of the idioms in the DSSA. The lead designers are then able to solve the recurring problems, generating canonical specifications that, in this case, update systems of objects and propagate the results of the update. A paradigm that separates objects,

connections, and the scheduling of updates seems effective. Since specifications using a DSSA are developed by the lead designers, the efforts of the lead designers are highly leveraged. The recurring problems are solved once and for all by the most talented people.

- ◆ A lean set of concepts is supported. The complexity of the DSSA is minimized. This allows for a more general specification applicable across simulator systems, i.e., the structure of the product is consistent and understandable. Consistency and understandability allow for efficient indoctrination of newcomers, education of maintainers, and confirmation of design decisions with users and program offices. Also, a quality assurance organization can be involved to assure that each system conforms to the DSSA. Thus, integrity of the product is assured.
- ◆ Design reviews now address design. Reviewers are able to discuss alternatives in the context of the DSSA. When the job is to pro-

² The comments stem from the use of a specific model for solving a specific problem and from the use of models in general. No attempt was made, however, to categorize the comments along these lines.

duce an instance of a general specification, which is optimal for the system being specified, reviewers can explore alternatives. Without constraints there are too many alternatives to explore. Tradeoffs are addressed at the right time and at the right level.

- ♦ Reuse is likely to improve using a DSSA. Reuse of design decisions is obvious. Less obvious is the observation that the software produced is more likely to be reusable in other systems using the same DSSA. The consistent structure of the specification enhances reuse of the software.
- ♦ The software should be easier to maintain. First, each system will be implemented similarly because the functional idioms are based on forms. Second, the separation of objects and connections isolates each system of objects. Thus, replacement or addition of systems affects only one end of a set of connections. The systems of objects at the other end of the connection do not need to change. Finally, aspects of a system, for example, an engine system, are simulated in identifiable localities; enhancements and simulations of malfunctions map to corresponding software entities.

4. Next Steps

Our intention is to discover and cultivate the notions of recurring specifications and DSSA. The hope is that the concepts will whet the appetites of others who will carry our ideas in other directions. Our aim then is to disseminate information about DSSA and accommodate the use of that information. So far, we have been able to do so in the context of the simulator domain, and we have begun to apply the technology to the command, control, communication, and intelligence (C²I) domain (see [3] for more information) and the embedded systems domain. We invite participation from those engaged in these and other domains.

Our work continues on several fronts:

- ♦ We are working with real-world program design teams to polish the descriptions of recurring problems, which constitute the bulk of a system.
- ♦ We are monitoring the use of DSSA on full-scale simulator developments to help mature the technology.

- ♦ We are producing a transition plan that addresses our interests in pursuing other simulation areas and other domains.

Our work on domain-specific software architectures has addressed the requirements of programs in the flight simulator domain. The DSSA approach involving recurring problems is a common-sense view of large software systems. The creation of canonical design specifications to recurring problems has provided practitioners with domain-specific examples. Such examples provide a focus for education, discussion, and evolution of the concepts of a DSSA.

Acknowledgments. This paper is based on work performed by the Domain-Specific Software Architecture Project at the Software Engineering Institute over the past two and a half years. The project members are Rich D'Ippolito, Ken Lee, Chuck Plinta, Mike Rissman, and Jeff Stewart.

References

- [1] Lee, K.J., Rissman, M.S. *An OOD Solution Example: A Flight Simulator Electrical System*. Technical Report, CMU/SEI-89-TR-5, Software Engineering Institute, Pittsburgh, PA, 1989.
- [2] Lee, K.J., Rissman, M.S., D'Ippolito, R., Plinta, C., Van Scoy, R. *An OOD Paradigm for Flight Simulators, 2nd Edition*. Technical Report, CMU/SEI-88-TR-30, Software Engineering Institute, Pittsburgh, PA, 1988.
- [3] Plinta, C., Lee, K.J., Rissman, M.S. *A Model Solution for C²I Message Translation and Validation*. Technical Report, CMU/SEI-89-TR-12, Software Engineering Institute, Pittsburgh, PA, 1989.

For more information contact:

*Kenneth J. Lee
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
Phone: 412-268-7702
ARPANET: kl@sei.cmu.edu*

The Role of Domain Independence in Promoting Software Reuse

Architectural Analysis of Systems

J.M.Perry, GTE Government Systems Corporation

M.Shaw, Carnegie Mellon University

Introduction

Domain analysis for reuse is a topic of much current interest and study. While there are several variations of domain analysis, they are usually characterized by their emphasis on application dependencies. This position paper describes architectural analysis which is a type of analysis for furthering our understanding of software architectures. It attempts to raise the abstraction level of design elements and, thereby, emphasizes domain independence. Although architectural analysis and domain analysis for reuse have different processes and goals, they are closely related and support one another. This mutual support is identified and examined. The SEI Software Architecture Project is described to provide an example of architectural analysis.

Domain Analysis for Reuse

Domain analysis is a discipline which is evolving and can be undertaken for different purposes. In the context of software reuse, several variations of domain analysis can be discerned. One type of domain analysis examines software systems in a well defined application area to identify operations, objects, and structures which are common to those systems. The common entities become candidates for construction as reusable software parts. The emphasis in this form of domain analysis is on application dependent parts for use in constructing software systems. A good example of this type of analysis is provided by [CAMP85]. Another type of domain analysis examines system requirements for a product to identify operational features which can characterize a product family (of systems). The emphasis in this form of analysis is on application behavioral characteristics for use in deriving system members of the product family. This type of analysis, features analysis, is discussed in [Perry88]. [Gish and Prieto-Diaz88] propose a general definition of domain analysis as the isolation, characterization, and abstraction for the purpose of creating domain taxonomies, models, and languages.

While these variations of domain analysis differ in their focus of attention or purpose, each attempts to understand an application area, or a family of application systems, by identifying domain specific characteristics which will lead to increased levels of software reuse for that application domain. There is an implicit assumption that the exploitation of domain dependencies will be more supportive of software reuse than sole reliance on domain independent artifacts; and that by concentrating on a relatively narrow domain, reusable artifacts will be more numerous and comprehensive if they are domain dependent. Consider, for example, the variety of reusable subsystems for avionics, including [Hitt83] navigation, guidance, display, steering, and [Parnas85] device interface, avionics data and physical model, and inertial measurement.

Domain Dependence and Domain Independence: Considerations for Reuse

While this focus on domain dependency is justified, it should be balanced with an appreciation for and understanding of the role of domain independence in moving software reuse from an ad hoc to systematic practice. The boundary between domain dependence and domain independence is subtle, often changing, and is one of degree. What often begins as a domain dependent artifact, through the right abstraction, can become applicable to another domain and, thereby, take on a degree of domain independence. For example, GTE Communication Systems Division evolved reusable telephone switching software[Roder78] and successfully applied the design of this software to other application domains, including flight controller trainers, C³ systems, and software engineering development environments. Another example of evolution from domain dependence to domain independence is provided by the message generation software[Lee, Plinta, Rissman89] developed by the Domain Specific Architectures Project at the SEI. This software originated within a C³ application to address the proliferation of message formats. The resulting software is reusable for other application domains, for generating, converting, and validating types of structures. Successful abstractions often begin as domain dependent concepts, which survive from ad hoc solutions to folklore practice, and through the suppression of some detail (while retaining the 'right' detail), become useful abstractions for systematic practice and eventual codification for application across several domains.

Study of descriptions of software systems indicates that some of the 'right' abstractions for supporting software reuse are design abstractions, pertaining to software architecture. The definition and formalization of these abstractions will not only promote the current practice of software reuse, but enable new kinds of reuse, at the architectural level. Domain analysis for reuse which focuses on domain dependent characteristics should be balanced by analysis which focuses on domain independent architectural abstractions.

Design Level Abstractions Enable Software Reuse

Abstractions enable programmers to handle program complexity. Higher level languages, abstract data types, procedures, and modules help programmers build 'better' programs. As the use of design abstractions, such as these, become widespread, canonical and specific instances are collected into libraries for reuse, and tools are developed to support their utilization in practice. Today, software problems involve system complexity and design abstractions for the system level are needed to help solve them.

The abstraction level of design determines the type and extent of software reuse which is possible and practical to achieve. Program statement abstraction raised the level at which reuse could be addressed from the machine statement level to the programming language construct level; and procedural and data abstraction raised the level of possible reuse from the programming construct level to a package and type level. Advancement to the next design level not only will enable reuse at a new higher level, but will lead to routine and systematic reuse at the prior levels.

Architectural Analysis for Higher Design Level Abstractions

The next advance will be to the system organization level and requires the identification of common architectural constructs and rules for their integration. Application domain analysis can support this advance if it has the goal of identifying important architectural elements in application system architectures and if the analysis does not rule out domain independent abstractions. This type of analysis is referred to, here, as architectural analysis. Analysis of these architectural elements and the ways in which they are combined can lead to good architectural abstractions for system organization. These abstractions can be used to specify architectural constructs, both domain dependent and domain independent ones, as well. If this can be achieved, software reuse based on these new constructs will be possible, thus, enabling more codification and systematic practice of reuse.

Architectural Analysis for Understanding System Organization and Reuse

Several basic approaches to software reuse include:

- identification of reusable components for development of a system;
- modification of a generic, base system to develop a new product;
- specification and invocation of the appropriate combination of primitives to elicit desired system behavior.

These three forms of software reuse are related, but differ with respect to emphasis on implementation-time or design-time; on lower or higher level of component; on parts or integrating framework.

The type of domain analysis for reuse should be determined by an explicit awareness of the chosen approach to reuse. For example, domain analysis for common parts is appropriate for 1); features analysis, for 2); domain analysis for a taxonomy [Prieto-Diaz86] was done for classification of work products in the context of 1). 1) is a constructive approach and 2) is a derivation approach. 3) is a generative approach and requires a deeper understanding of system organization.

Architectural analysis can support reuse, including generative reuse, by focusing attention on higher design levels and system frameworks. It will increase our understanding of the composition of systems and the relationship of system organization to system behavior.

System organizations are constructed from subsystems and composition mechanisms. The issues of system construction are: the nature of the subsystems, function of the subsystems, internal structure of the subsystems, the composition or integration mechanisms, and subsystem operation and behavior. Each of these may be dependent on the implementation of a specific system, a family of closely related systems, or to many systems. Identifying and classifying the system functions that are common to a domain is a start in addressing these issues. Architectural analysis then extends this identification and classification across domains in order to further resolve these architectural issues. The ultimate objective of reuse is construction of 'better' systems more productively. Understanding these architectural issues will provide the foundation for reuse to achieve its objective.

In order to understand reuse for a domain, it is necessary to understand the architecture for systems of

that domain, to identify the 'right' design abstractions which involve a relationship between domain independence and domain dependent characteristics.

SEI Architecture Studies

Several SEI projects are addressing software architectures. The SEI Software Architecture Project [Shaw88], also known as the Vitruvius Project¹ is investigating application independent architectural elements and principles for describing, analyzing, and constructing software systems. The development of a theory of architectural design involves structures, specifications, virtual machine issues, design abstraction, and rules combining system types, and implementation choices.

The plan of the project consists of two parts. First, a breadth-first analysis of system descriptions, across domains, will be conducted to accumulate examples of subsystems and system organizations. These will be categorized to identify architectural abstractions useful in describing system architectures. Second, an in depth analysis of specific systems will be conducted to refine the architectural elements and principles, implementation alternatives, comparisons, and criteria for architectural decisions.

The project is currently producing a survey of systems and software architectures. This survey has identified abstract data type[Liskov87], pipes/filter, layers[Oberndorf88], client/server[Spector87], central repository[Erman80], and dependent processes[Barbacci88][Rosene81] as initial candidates for architectural elements.

Another SEI project, DSSA² project [Rissman89] [Lee89], is taking a complementary approach to software architectures. It is investigating specific domains to identify characteristic or recurring problems of those domains and, then, formulating canonical solutions to those problems. The canonical solutions are represented as solution patterns for reuse in those domains.

Domain Analysis Provides Architectural Insights

The various domain analyses for reuse provide insight into the influence of domain dependent characteristics on system organization. Architectural analysis will provide constructs in the form of domain independent subsystem types, along with their integration mechanisms. The implementation of these types in a specific application domain involves design and implementation choices which incorporate more detail, including domain dependent characteristics. Typical choices involve data structure representation, packaging or grouping of subsystems, distribution of control, performance and resource usage, and allocation to hardware. Understanding architecture includes not only architectural elements and interconnection mechanisms, but implementation alternatives and implications for use in a particular domain. Example of alternatives and their impact are provided by various domain analyses. The formulation of the 'right' architectural abstractions and their usefulness to build 'better' systems requires the insights which the domain analyses for can provide. Thus, while domain analysis for reuse and architectural analysis differ in their emphasis on domain dependence and independence, they are

¹Marcus Vitruvius was the Roman architect, active from 46 to 30 BC, who wrote the only surviving work, a ten volume treatise, on ancient architectural theory and practice.

²Domain Specific Software Architectures

mutually supportive, from different approaches, of the same goal, namely, building 'better' systems.

Conclusion

Architectural analysis complements other types of domain analysis for reuse by providing a perspective on higher design levels which enable systematic reuse; a perspective which encompasses both domain independent and domain dependent characteristics.

On the other hand, domain analysis for reuse supports a study of architecture by providing understanding of domain characteristics necessary for applying architectural abstractions to a domain. For software architectures, it is not sufficient to just identify system structures, subsystem types, and techniques for composing subsystems into systems. These must be accompanied by implementation details about alternatives, comparisons, tradeoffs, and application criteria. Understanding reuse for a domain will help obtain this knowledge.

Architectural analysis helps us build 'better' systems more productively. Architectural abstractions will help reduce the complexity of systems, improve their reliability through well understood subsystems and integration rules, and increase productivity of development and maintenance by enabling more software reuse.

BIBLIOGRAPHY

Barbacci, M.R., Weinstock, C.B., and Wing, J.M. Programming at the Processor-Memory-Switch Level. In Proceedings of the 10th International Conference on Software Engineering, April, 1988.

CAMP - Common Ada Missile Packages. Final Technical Report, Vole. 1, Overview and Commonality Study. Results, McDonnell Douglas Astronautics Co., September, 1985.

Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Reddy, D. Raj. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. Computing Surveys 12(2):213-253, June, 1980.

Gish, J., Prieto-Diaz, R. Domain Analysis: Procedural Model Refinement and Experiment Proposal. GTE Laboratories, April, 1988.

Hayes-Roth, Frederick. Rule-Based Systems. Communications of the ACM 28(9):921-932, September, 1985.

Hitt, E.F., Kluse, M., and Broderson, R. A Core Software Concept for Integrated Control. Journal of Guidance, Control, and Dynamics 6(3):215-217, May-June, 1983.

Lee, K., Pflint, C., Rissman, M. Domain Specific Architecture Report. To be published in 1989.

Liskov, Barbara. Data Abstraction and Hierarchy. In OOPSLA '87 Addendum to the Proceedings, pages 17-34. The Association for Computing Machinery, New York, NY, October, 1987.

Oberndorf, Patricia A. The Common Ada Programming Support Environment(APSE) Interface Set(CAIS). IEEE Transactions on Software Engineering 14(6):742-748, June, 1988.

Parnas, David L, Clements, Paul C., and Weiss, David M. The Modular Structure of Complex Systems. IEEE Transactions on software Engineering SE-11(3):259-266, March, 1985.

Perry, J. Perspective on Software Reuse. Technical Report, CMU/SEI-88-TR-22, Software Engineering Institute, Pittsburgh, PA.

Prieto-Diaz, R. Domain Analysis for Reusability. GTE Laboratories, Waltham, MA, December, 1986.

Rissman, M. GTE Government Systems Colloquium Presentation. GTE C³ Systems, Needham, MA, March 1989.

Roder, J. Phoenix Architecture. SIGDA Newsletter 8(2):18-22, June, 1978.

Rosene, A.F., Connolly, J.E., Bracy, K.M. Software Maintainability: What It Means and How to Achieve It. IEEE Transactions of Reliability, 30(3), August, 1981.

Shaw, M. Toward Higher-Level Abstractions for Software Systems. Proceedings of the Third International Symposium on Knowledge Engineering, Madrid, SPAIN, October 1988.

Spector, A.Z., et.al. Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report. Technical Report TR CMU-CS-87-129, Carnegie Mellon University, June, 1987.



Position Paper

Software Reuse

Chris Taylor
Software Consultant
Airborne Display Division
GEC Avionics Ltd., Rochester, England.

The Airborne Display Division (ADD) of GEC Avionics Ltd. manufactures display systems for aircraft, using some of the most advanced technology in the field. Their primary product is Head Up Display Systems, in which the company are world market leaders, supplying systems for most aircraft types including all variants of the General Dynamics F-16 Fighting Falcon. Related products include Head-Down and Helmet-Mounted Displays. The author acts as a consultant within ADD, advising on systems development and the efficiency of the software development process.

These are the views of the author and do not necessarily represent those of GEC.

1: Software Component Technology - Management Issues

1.1

Procurement Policy

There are two aspects of government procurement policy which particularly influence defense contractors developing software: the ownership and usage stipulations of Federal Acquisition Regulation supplement 252.227-7013 (Rights in Technical Data and Computer Software), and the development practices required by DoD-STD-2167A (Defense System Software Development).

A software development organization has an incentive to produce reusable components only if the organization retains the right to profit from any future use of those components. Where an

organization develops software in order to add value to its primary product, software components are frequently developed within the meaning of the term "Developed Exclusively with Government Funds" as defined by the cited FAR. That being the case, the government generally enjoys unlimited rights in the software produced. It is therefore questionable whether the organization could legitimately charge for any future use of that component, and indeed whether the organization is then free to use that component in products not intended for the U.S. government. In this respect, current procurement policy provides little incentive to organizations whose primary product is not software.

Software developed for the government is generally prepared in accordance with DoD-STD-2167A. Although the practices detailed in this standard do not preclude extensive use of software components, the lifecycle model and data requirements provide little support for or active encouragement of software component technology.



1.2

Costs of Component Technology

The marginal cost of developing a reusable software component (compared with a "custom" component of similar functionality) is largely associated with the work necessary to ensure correct operation of the component under all conditions. Where the operating environment of a component is completely specified, as it generally is for custom components, the manufacture and testing of the component can be optimised for that environment. Reusable components must be designed to operate in a variety of environments, and testing of the component must be extended to ensure correct operation within environments other than that of the product for which the component was originally required. The more general the component, the more costly this extra design and testing becomes. Software to be used in real-time embedded systems cannot be subject to incorrect or suboptimal performance, since in many cases mission effectiveness or operator safety are affected. Current tools for the development of software components do not provide sufficient support to allow certification of components.

A reusable software component is an asset of the organization responsible for its development, and the decision as to whether a particular reusable component is to be developed should therefore be approached in the same manner as any other capital project. The use of the payback period for this evaluation is inappropriate, since it ignores the time value of money and encourages short-termism. Discounted cash flow methods should be used for the evaluation. In particular, the *estimated net present value* of the software component should be determined. This has the advantage of focussing attention on the cost of capital for the project. Since the development of reusable components often requires the services of scarce resources such as experienced design engineers, the determination of the appropriate discount rate for the project can reflect the impact of the development project on the capacity of the developing organization.

Since software is not the primary product of ADD, the organization has not established an economic incentive to develop a repository of

reusable software components. Such components themselves have no intrinsic value, since software serves only to add value to deliverable hardware systems. The concept of software reusability presupposes that software components are expensive to fabricate. Where custom software can be developed at negligible cost, perhaps by automatic code generation from a suitably formal requirements specification, the maintenance of a code repository becomes unnecessary. Experiments with a rule-based automated code generator have been carried out with some success. However, this technology cannot be applied as yet to complex components.

2: Domain Analysis - Avionics Displays

2.1

Domain Analysis

Within the overall domain of software developed for avionics displays, there are a number of identifiable subdomains which can be used to categorise software components.

2.1.1 Operating System Subdomain

Embedded systems application software does not normally enjoy the services of an operating system such as UNIX. Even where software is being developed using the Ada language, a layer of software must be constructed which provides framing or scheduling for other components, and performs periodic built-in test of the underlying processing resources. Since new products are often based on a previously developed hardware architecture, reusable operating system components can be developed which provide this functionality.

2.1.2 Common Services Subdomain

All Displays product software shares the need for common services such as mathematical routines, digital filters, data validation and reversion services, coordinate frame transformations and so on.

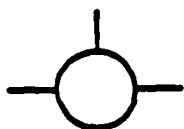


2.1.3 Communication Services Subdomain

Avionic display systems are in general "information sink" systems. The function of a display system is to integrate information from a number of disparate aircraft systems and present it to the aircrew. Most of these aircraft systems have well defined interfaces, some being defined by standard vendor-independent functional interfaces (e.g. Inertial Navigation Systems and Air Data Computers). Reusable software components can therefore be developed to provide "protocol stacks" which implement interfaces to these systems.

2.1.4 Display Format Subdomain

The pictorial format used to present information to aircrew is determined by the customer. Most formats are in fact very similar, irrespective of the particular airframe for which the system is being developed. Software components can therefore be developed which provide object-oriented implementations of specific symbols. For example, the following symbol is widely used to represent the aircraft velocity vector (where the aircraft is going):



This pictorial "object" can be moved on the display screen via the Cartesian coordinates of the center of the circle. In addition, the symbol can normally be either displayed or not displayed. A reusable software component can therefore provide this display service to the application software.

2.1.5 Display Mechanization Subdomain

In some cases, the information necessary to manipulate display symbols is provided directly by external systems. However, it is often the case that the display system itself must generate the symbology control information from more basic data. Standard solutions exist for many of these mechanizations. For example, the cartesian coordinates of the velocity vector symbol illustrated

above can be derived from aircraft attitude and velocity data.

Basic avionics display systems can be constructed from combinations of the above components, along with sufficient "custom" software to provide the display behaviour required by a specific customer. Most display products in fact contain additional capabilities, such as navigational aids and the computation of weapon impact points.

3: Reuse in Action

3.1

The AdaHUD Program

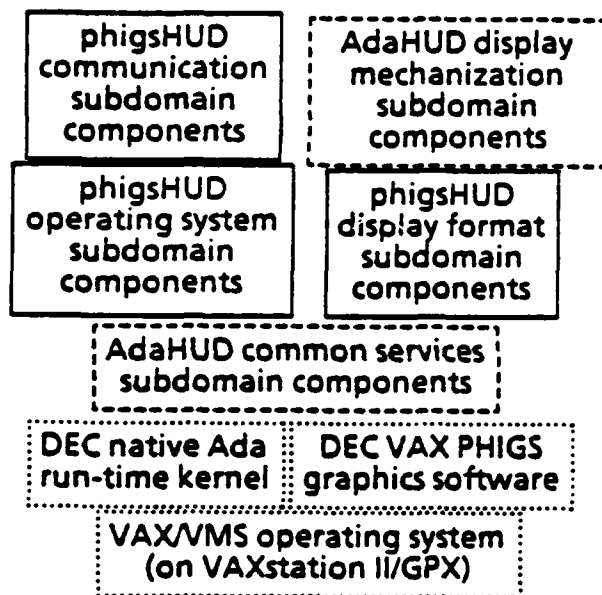
In September 1988 ADD demonstrated a Head Up Display (HUD) system at the Farnborough Airshow in England. This private venture project required the development of a software system which would reproduce European Fastjet standard symbology using unmodified F-16 C/D HUD hardware. The software was developed entirely in Ada, using the InterACT compiler targetted to the MIL-STD-1750A processor. All the features of the Ada language were used in this development, including tasking, instantiations of generic components, and exceptions. The successful completion of this exercise demonstrated the feasibility of Ada for avionics display purposes.

The development of a new display system (phigsHUD), which would implement the ANSI standard graphics language PHIGS was also under way. The implementation was entirely in Ada. In order to investigate the utilisation of PHIGS for avionics displays, the development of a prototype was targetted to a DEC VAXstation[†], using the DEC VAX PHIGS software. Components of the AdaHUD software system were recompiled using the DEC

[†] VAX and VMS are registered trade marks of the Digital Equipment Corporation (DEC)



VAX native Ada compiler, with new components being generated for the display format subdomain which made use of PHIGS commands instead of the proprietary graphics language used on the original AdaHUD/F-16 development. Similarly, new operating system components were developed to allow the new application to execute above VAX/VMS. Finally, "communications subdomain" components were developed to simulate the activities of external avionics systems. Thus, the PHIGS system prototype was constructed from components taken from the AdaHUD program, with new components being added where necessary, as shown in the following diagram:



This approach was entirely successful, and demonstrated the power of reusable components in systems development. Since the phigsHUD system was simply a prototype, the concerns over component certification were not applicable. At the time of writing, the deliverable phigsHUD software is in development.

4: Summary

4.1

Software Reuse in Display Products

The most active area of reusable software component development at the present time is the generation of components which can be used a number of times *within a single project*. There are essentially two reasons why this is the case. Firstly, an economic incentive exists for the project manager to develop reusable components which reduce the total development cost of a product. Since there is no attempt to justify investment beyond the time horizon of the project, discounted cash flow becomes irrelevant and the decision as to whether or not to make a component reusable becomes simply that of ensuring payback within the lifecycle of the product. Secondly, the external environment of the components is well known, which therefore reduces the costs of designing and testing the components.

In order for more generally reusable components to be developed, the legal aspects of rights in reusable software must be addressed, and military standard software development practices must encourage non-waterfall lifecycle models. The costs of generating custom software components must then be higher than the cost of developing and maintaining reusable software components for the technology to become attractive.



The Charles Stark Draper Laboratory, Inc.

555 Technology Square, Cambridge, Massachusetts 02139
Mail Station 3A

Telephone (617) 258-2747
LHC2747@DRAPER.COM

May 19, 1989

Position Paper: Reuse in Practice Workshop

Pittsburgh, PA July 11-13, 1989

Submitted by Leigh Anne Clevenger, CSDL Ada Office

Introduction

Can reuse really cut software development costs? Can significant amounts of software be reused in a real project? How will design for reuse be enforced?

These are the types of questions we get when discussing Ada software reuse. Since reuse and maintainability were important factors in the original adoption of Ada, answers to these questions should exist. The fact is that definite answers to these and other reuse questions are not yet available.

The Ada Office at the Charles Stark Draper Laboratory, Inc. (CSDL) has been looking into reuse issues for the past few years. In particular, our inquiries have concentrated on reuse of software designed to be run on embedded systems, useful library taxonomies, and quality evaluation of reusable components. This position paper describes a reuse task for NASA we are currently performing, our in-house Ada repository, and some other ways in which CSDL is participating in Ada reuse.

NASA Reuse Task: "Reusable Software Flight Certification Requirements"

The Reusable Software Flight Certification Requirements task is being performed by the CSDL Ada Office for the National Aeronautics and Space Administration (NASA) Level II office in Reston, VA. The task is administered through Johnson Space Center in Houston, TX. We began on March 1, 1989, submitted the first part of our report on May 1, and will complete this phase of the task on September 29, 1989. We anticipate the task will continue for 2 more years. The following is an introduction to the task, the task description and approach, and some questions we were asked at the first task review at JSC.

Task Introduction

The use of reusable Ada software components in the development of flight software has the potential of providing major cost savings to the Space Station "Freedom" Program (SSFP). NASA's Software Support Environment (SSE) will provide a library of such reusable Ada components which will be obtained from software developed by SSFP prime contractors and from other non-NASA Ada software development organizations. However, the use of reusable Ada software components in SSFP flight systems requires that the components be flight qualified prior to their incorporation into the library. This study will identify flight software qualification criteria and the associated process that must be implemented in order to qualify reusable Ada software components for use in onboard application software.

Task Description

The task will perform the following and document the results:

1. Describe how qualification of reusable Ada flight software fits into plans for the Software Support Environment and the Space Station "Freedom" Program.
2. Develop a base set of requirements, assumptions, and quality criteria applicable to reusable Ada flight software.
3. Determine the quality attributes (criteria) that reusable Ada flight software should have for entry into the Software Support Environment reuse library as Flight Certified Ada Parts (FCAPs).
4. Define "classes" of flight software that will group reusable components based on an evaluation of their overall reliability and performance.
5. Define the process that will be used to apply qualification criteria to potentially reusable Ada software.
6. Identify and prioritize candidate components for the qualification process.
7. Apply the qualification process to representative software components from the list of suitable candidates.
8. Identify tools and future work needed in the area of flight qualification of reusable Ada software units.

Task Approach

In order to achieve our goal we intend to use, as a reference base, surveys of existing software qualification criteria, current Ada repository software qualification procedures, quality requirements proposed for reusable Ada components in technical literature and reports, and discussions with flight software developers and quality assurance specialists.

Initially, "classes" of flight software will be defined. This will enable the grading of each reusable component based on the criticality of its application within the SSFP. All reusable Ada parts designated for use in the same class of an application will have to be certified to at least that class level. Higher levels of certification will require a systematic progression of more stringent criteria than lower levels of certification. For each class of application, we will determine the attributes that Flight Certified Ada Parts (FCAPs) should have. This will entail the identification of quality and other criteria for flight software. The process used to apply the criteria to candidate components within the Software Support Environment Development Facility will then be defined. Representative software components will be identified and prioritized for testing the process model. In this way, the process model will be verified and validated.

Questions Raised

The following are examples of questions raised during our first review of this task:

What is the difference between "Common" software and "Reusable" software?
Common software is the term for large collections of software which are developed to be used intact by more than one segment of a project.

Should we attempt to qualify other software products such as requirements, designs, etc., or just Ada source code? Should we only qualify software products written with reuse in mind?

What assumptions can be made during the evaluation of the quality of reusable Ada components?

What is the estimated reduction in cost and time which reuse will provide?

Who will make people reuse software? What about possible cost incentives for reuse? A special effort to follow up on reuse practices is needed.

CSDL Ada Repository

In looking out for CSDL's own reuse needs, last fall we developed a prototype Ada repository, initially populated with the CAMP parts and some reliable CSDL-developed components. This database is on-line for use by any Ada programmers and designers at CSDL.

The issues addressed in developing the repository included taxonomy definition and automated database construction. The taxonomy for the CSDL repository was developed with the embedded systems domain in mind. We were also limited by our library management system to a fixed database entry structure. As this is purely a prototype effort we didn't feel justified in purchasing a new database system, and instead utilized an in-house version of IBM's Info database. Searches are performed on keywords entered by the user. Since the CAMP Ada source code is written in a standard format, we developed an automated process for getting source code attributes into the database entries.

Other Reuse at CSDL

The CSDL Ada Office has submitted a proposal to the Air Force to write a methodology to guide the development of maintainable avionics software for advanced avionics architectures, and to develop a second methodology to direct maintenance and support activities including effective use of Ada reusable components. We are frequent contributors to seminar and conference sessions on reuse, most recently the SIGAda Reuse Working Group meetings at Tri-Ada in October 1988, the SIGAda meeting in California this spring, and the SEI 1989 Affiliates Symposium.

In all the work we have done on reuse, people are enthusiastic at the prospects of cost and time savings, but want evidence that it works before they make the investment. By attending the Reuse in Practice Workshop we hope to be able to focus on implementation issues with others involved in reuse efforts. We also welcome the chance to share our research experiences with others.

Ada Office

A core group of computer engineers is supported by CSDL to examine important issues related to Ada and software engineering. Members of the "Ada Office" are currently participating in projects ranging from compiler and tool evaluation to full life-cycle software engineering to implementation of hard-deadline real-time embedded systems. Ada Office personnel include Brooke Green, Anne Clough, Sidney David, and Leigh Anne Clevenger.

Towards a design philosophy for reuse.

E.M. Dusink
T.U. Delft
Faculty of technical mathematics and informatics
Julianalaan 132
2628 BL Delft
the Netherlands

1. Integration of reuse aspects in the reuse process

As a necessary precondition for reuse to happen, the following topics of reuse should be considered:

1. the type(s) of reuse, building block approach or transformational approach,
2. the level(s) of reuse, code or design,
3. a design method which fits with the kind(s) and level(s) of reuse wanted,
4. tools which support the design method,
5. support for the actual construction of software.

Some of the topics are addressed in literature. However, no integration of the topics into a single framework was found. Our goal is to present a reuse framework in which all topics are considered and addressed in a coherent way.

To reach this goal a project on reuse was started at the T.U. Delft, in which was chosen for the building block approach, including white box and black box reuse. In this project we want to establish both reuse on source-code level and reuse on design level. By choosing these approaches new questions arose: what has to be the components form and should be its interface look. The question about the actual construction of software has to be transformed into a question how to connect the components.

Our own interest in reuse started from an Ada background [Dusink], we built our own Ada compiler [van Katwijk], we did a study to transform Algol 60 to Ada automatically [Huijsman], and we give courses on Ada and software engineering. Both the choice for components based reuse (building block approach) as well as the choice for an object-oriented design method were influenced by this background.

In this paper we address: a reuse-based design method in which our ideas about components are incorporated, design supporting tools, and we present a short description of our reuse project.

2. A software design method oriented towards reuse

Design methods reported in literature are seldom tailored towards reuse, although object-oriented design methods claim to have reuse as a side effect. (In [Deutsch] even three forms of reuse are claimed for object-oriented languages.) However, in none of these articles the design method is put in a framework.

We based our design method on the observation that experienced application domain programmers work with a set of mental primitives of the application domain. (In our case, the application domain is systems programming, and the reused components are related to the UNIX system routines.) The importance of this domain knowledge is shown by a.o. [Levy]. We concluded that our design method had to be domain oriented or should cover the process of the acquisition of domain knowledge. We chose for incorporating a method of acquisition of domain knowledge. After all, methods for acquisition of domain knowledge are less sensitive to changes in the application domain. In this way the resulting design can easily be mapped to existing components.

Apart from being oriented towards actually reusing software, the design method should be such that new pieces of reusable software are a result as well. According to the literature mentioned, an object-oriented approach should do the trick.

The kind of the stored components has to be compatible with the kind looked for during the design process and with the kind delivered by the design method. In this way unnecessary transformations are avoided. Already existing reusable code should not be excluded from reuse by the design method. As existing code has different forms, from algorithms via abstract data types to abstract machines, a small problem rose. But, as in object-oriented design all three forms are manipulated it turned out to be a non-problem.

For our design method we assume the existence of a repository. The global outline of our design method, in which we considered the issues mentioned in [Ladden], is as follows:

Step 1: A software requirements document is made.

Step 2: Orientation on the application domain as a whole, browsing of the repository, try to get an idea about the usual primitives/blueprints common for the application area.

Step 3: First, the entities on the top level are looked at. These are the problem-space entities.

1. An informal strategy.

The top-level entities are defined, with their functional primitives. Then the repository is queried to see if there are useful components. Probably a rearrangement of entities and their functional primitives over objects is necessary or other objects have to be chosen.

2. Formalize the strategy

The components found in step 3.1 are mapped on the defined objects.

The result of step 3 can be one or more of the following:

1. designed objects for which components from the repository with the same specification exist. If more than one component exists for an object, one will be selected at a later stage.
2. objects for which components from the repository with almost the same specification exist, but some tailoring is needed. If more components exist for one object, one will be selected at a later stage.
3. objects for which there are no components in the repository with the desired specification. These objects have to be implemented later.

Step 4: Detailed design. This step is essentially the same as step 3. However, now a complete architecture is designed.

1. Develop an informal strategy. A complete architecture is defined, thus the problem-space entities plus the necessary solution-space entities with their functional primitives and their interconnections. The repository is queried to find matching components.
2. Formalize the strategy. A formal search based on the more formal specification of the objects that is the result of the former steps is done.

Step 5: Implementation and testing. The objects for which no components were found in the repository are implemented. If the transformation step from class description to source code is too large a decomposition of these objects can be done according to step 3 and 4. If in step 3 and 4 more than one component was found in the repository a choice has to be made now. Furthermore, tailoring needed for the components mentioned in step 3.2 is done.

Criteria to be used in the forming of classes are:

- minimizing the number of connections among classes
- minimizing the flow of information among classes
- getting logically coherent classes

A rationale for all three criteria is that they determine the ease with which the architecture of the system can be understood as well as the ease with which the functionality of a class can be understood.

A strong precondition for the design method to work is that the stored components have to reflect at least the basic primitives of the application domain(s). This can be obtained by surveying existing software, detecting objects in it and extracting them. Several systems on the same application domain have to be viewed and similarities marked down. The objects detected have to be made more general before putting them as reusable components into the repository. It must be recalled that making components more general does not imply changing them into reusable components. The ease of reuse can become less by complex interfaces, etc.

3. Design supporting tools

The terminology used in a repository and its associated facilities should be compatible with the terminology used in the design method. The facilities should also support the design method. Our support tools were derived from the design method. In the following we concentrate on tools working on the repository.

The following tools are needed:

- a browser to support step 2 of the methodology.
- a faceted scheme query system (similar to the one proposed by [Prieto-Diaz]) to support step 3.
- a related system, with which related components to the ones already found can be given, to support step 2 and step 3.
- a formal specification system is needed to support step 4. This system has to work on the repository as a whole as well as on an already selected set of components.
- tools to allow inspection of components, for both the informal as well as the formal strategy.
- tools to support tailoring.

We believe that the repository, together with its facilities, should not form a small isolated environment but should be integrated in a software engineering environment.

The components can be stored in one or several repositories according to the application domains. Component distribution over repositories is transparent to the user, unless the user asks for a special application domain.

We do not use the ideas of Prieto-Diaz of using conceptual closeness and fuzzy logic to give an ordering of found components as a help in choosing the best component. This ordering depends too heavily on the users' ability to give weights to closeness, it is therefore very liable to give only a false impression of helpfulness.

Procedures should be established to guarantee the quality of the components in the repository. Our solution is to have one central repository with a responsible librarian. Of course, such an approach is based on company policies.

4. Our experience with reuse

At Delft, a PhD student is evaluating the design method by applying it at system software. A repository, together with its facilities, was designed according to the statements mentioned before. It is currently being prototyped by some students. Apart from evaluating the design method and prototyping the repository, guidelines about the appearance of components are made.

This is also done by a PhD student. These guidelines are being evaluated together with the further evaluation of the design method.

Points of research at this moment are a module interconnection language that can be used with the repository and the reuse of designs.

5. References

- [Deutsch] Deutsch, L.P. (1983) Reusability in the Smalltalk-80 Programming System. Proc. of the workshop on reusability in programming Newport, RI, September 7-9, 1983
- [Dusink] Dusink, E.M., Katwijk, J. van (1987) Reflections on reusable software and software components. In: Ada components: libraries and tools. Proc. Ada-Europe International Conference, May 1987, pp. 113-126 Stockholm 26-28 May 1987, Ed. S. Tafvelin, The Ada Companion Series, Cambridge University Press 1987
- [Huijsman] Huijsman, R.D., Katwijk, J. van, Pronk, C., Toetenel, W.J. (1987) Translating Algol 60 programs into Ada: Report on a feasibility study. Ada Letters V 7 (5), pp. 42-50, September/October 1987
- [van-Katwijk] Katwijk, J. van (1987) The Ada- compiler: On the design and implementation of an Ada compiler. PhD Thesis, TU Delft, the Netherlands
- [Ladden] Ladden, R.M. (1988) A survey of issues to be considered in the development of an object-oriented development methodology for Ada. ACM Sigsoft Software Engineering Notes V 13 (3), pp. 24-30, July 1988.
- [Levy] Levy, P., Ripken, K. (1987) Experience in constructing Ada programs from non-trivial reusable modules. In: Ada components: libraries and tools. Proc. Ada-Europe International Conference, May 1987, pp. 100-112 Stockholm 26-28 May 1987, Ed. S. Tafvelin, The Ada Companion Series, Cambridge University Press 1987
- [Prieto-Diaz] Prieto-Diaz, R., Freeman, P. (1987) Classifying Software for Reusability. IEEE Software V 4 (1), pp. 6-16, January 1987

Ada and RESOLVE: Toward More Reusable Ada Components

Stephen Edwards
Institute for Defense Analyses

The current interest in software reuse had led to the reexamination of modern programming languages. Ada, the computer language adopted by the Department of Defense, has come to the fore under this issue, especially because of the DoD's interest in software reuse. As a result, weaknesses in Ada in this regard have been observed [Gargaro 87, Muralidharan 89]. Many research efforts are trying to address these problems, either through proposed Ada 9X changes or through completely new languages. Unfortunately, Ada revisions may be restricted in scope to maintain backward compatibility with current Ada code and to avoid drastic revisions to current compilers, while new language efforts are discounted because of the "language in a vacuum" problem—incompatibility with software written in current languages and little tool support.

A possible answer to this dilemma is to create a new reuse-oriented language with a compiler that produces target code in a commonly used high-level programming language, such as Ada. If it is possible to ensure that all of the benefits gained by using the new language are captured in the target language representation, then it is possible to have the best of both worlds—freedom from current language restrictions while maintaining compatibility with the current software base.

In pursuit of this idea, the language RESOLVE, currently under development by Bruce Weide at Ohio State University, seems to be a good candidate [Harms 89a, Harms 89b]. This language is aimed at providing the following:

1. A complete encapsulation mechanism,
2. A mechanism for the efficient implementation of all language features,
3. Semantic, as well as syntactic, specifications,
4. The capability of multiple implementations per specification, and
5. Verifiability of implementations against the semantic specification.

These points address the major concerns about the use of languages such as Ada for writing reusable software. RESOLVE is designed around a programming model which is very different from that used by more traditional computer languages and which encourages the exploitation of these capabilities for producing reusable software.

To demonstrate how RESOLVE addresses these issues, consider an example program unit which is a candidate for reuse. The unit chosen for this example is a generic stack module, often used in text books to demonstrate modular design. Figure 1 shows a

straightforward Ada specification for this unit.

generic

type T **is** private;

package Bounded_Stack_Template **is**

type stack **is** private;

function new_stack (max_size : **in** integer) **return** stack;

 -- for initializing at declaration

procedure set_max_size (s : **in out** stack; max_size : **in** integer);

 -- for dynamically resizing

function get_max_size (s : **in** stack) **return** integer;

function get_size (s : **in** stack) **return** integer;

procedure push (s : **in out** stack; x : **in** T);

 -- raises STACK_ERROR when s is full

procedure pop (s : **in out** stack);

 -- raises STACK_ERROR when s is empty

procedure pop (s : **in out** stack; x : **out** T);

 -- raises STACK_ERROR when s is empty

function top (s : **in** stack) **return** T;

 -- raises STACK_ERROR when s is empty

function isempty (s : **in** stack) **return** boolean;

function isfull (s : **in** stack) **return** boolean;

procedure free_stack (s : **in out** stack);

 -- for reclaiming space

 stack_error : exception;

private

type real_stack_type **is** array(positive range \diamond) of T;

type stack **is** access real_stack_type;

end Bounded_Stack_Template;

Figure 1—Bounded_Stack_Template specification in Ada.

Although this seems to be a very reusable Ada generic, in practice it may often be unsuitable. It fails to meet all five of the criteria above.

Complete Encapsulation: Initialization and Finalization

The *Bounded_Stack_Template* package suffers from inadequate encapsulation because it cannot enforce the initialization or finalization of objects. First, the type *stack* is not adequately encapsulated. Although there is provision for initializing this type at its declaration with *new_stack* (or later, with *set_max_size*), the author of this unit cannot

enforce its initialization and, therefore, cannot be assured that all objects of type *stack* start out with safe values. Likewise, there is no way the author can enforce finalization via the *free_stack* routine. In some cases that may only mean space is not reclaimed, but in other cases (such as data structures maintained with associated reference counts), internal data structures may require finalization. Second, the parameter data type *T* is not sufficiently encapsulated. The only operations defined for objects of type *T* are assignment and equality comparison. There is no way for the package *Bounded_Stack_Template* to ensure correct initialization of any objects of type *T* (although stacks may be implemented safely without this capability). In addition, when a non-empty stack is destroyed, via *free_stack* or *set_max_size*, there is no way for the package to finalize the remaining elements. There is also no way to finalize stack objects (or stack contents) when they are overwritten by values returned from *new_stack*.

In addition to the above restrictions, notice that only types for which the builtin assignment operator is appropriate can be passed into this generic. Because of the semantics of *push* and *top*, the assignment operator is used to make copies of objects of type *T*. If *T* were an access type which was supposed to represent a complex data structure like a binary tree, then a separate *copy* function would also have to be passed into the generic in order for the correct semantics to be implemented (alternatively, the semantics of the procedures could be altered to eliminate copying in this instance). Furthermore, there is no copy operation supplied by this package for the type *stack*, prohibiting the use of this type as the parameter for any further generics which require true copy semantics, since assignment on objects of *stack* type would not suffice. Thus, a stack of stacks cannot be created with this specification (and provide the appropriate semantics).

Efficient Implementation

The main limit to implementation efficiency in this unit is the copy semantics associated with the *push* and *top* routines. *Push* places a copy of its input parameter *x* on the stack *s*, and *top* returns a copy of the top value of the stack *s*. The copy operation is inherently linear in the size of type *T*, and thus nothing about the performance of the package *Bounded_Stack_Template* can be said independently of its instantiation parameters. In addition, the cost of true copy semantics for large data structures will be too high for this unit to be reused for such structures. This problem may be remedied by either removing the copy semantics, or providing two sets of *push* and *pop* routines—one with copy semantics, and one with produce/consume semantics.

Semantic Specifications

Since Ada specifications are purely syntactic, it is clear that no semantic behavior is specified in this unit (beyond the parameter modes). However, a full description of the behavior is necessary in order for a component user to understand and utilize such a reusable unit. For a well-known abstraction like *stack*, this is not a significant problem, but for less common abstractions, it is vital for a behavioral description to be available. If the behavioral description is formal, then it is not only less ambiguous for the user, but tools may check that the user is actually employing the abstraction correctly (i. e., meeting the preconditions). In addition, semantic specifications are necessary to provide

verifiability, which is discussed below.

Multiple Implementations

This Ada specification precludes multiple implementations for the same abstraction. In some cases, it may be more efficient to use an array for a stack, such as when indexed accessing is required. In others, a different implementation may be useful, such as using a linked list when the size fluctuates dramatically and often. Having a single specification and many implementations for such a unit is desirable, especially when the implementation can be chosen at instantiation time. The declaration of the type *real_stack_type* in the private part of this specification could be placed in the body, allowing multiple implementations to be written for this unit. Unfortunately, at elaboration time Ada allows only one body for each specification. If there are multiple implementations for a unit, only one may be chosen for all instantiations. Even if multiple implementations are provided, only one can exist at elaboration time. Thus, a compilation unit could not instantiate this package multiple times, using different implementations as appropriate. There is currently no workaround for this in Ada, other than having a separate specification for each implementation.

Verifiability

The problems of verifying Ada code are well known. In addition, the lack of semantic specifications does not provide anything to verify the implementations against. Lack of verification does not prevent reuse, but does raise its cost. This is due to the fact that it is less desirable to reuse code of unknown quality, and is more costly to debug a project when the bugs cannot be isolated to only new code. Semantic specifications and implementations which are verified against them would help in alleviating this problem. Unfortunately, the cost of developing these tools for Ada may actually outweigh the advantages.

A RESOLVE Example

Figure 2 shows the same *Bounded_Stack_Template* in the language RESOLVE. It addresses each of the shortcomings illustrated in the Ada example.


```
module Bounded_Stack_Template (T : type)
  theories
    string_theory, numbers
  provides
    type stack is (items : string(T), max_size : integer) = ( $\Lambda$ , 0)  (* initially empty *)
    procedure set_max_size (alters s : stack, preserves max_size : integer)
      requires max_size > 0
      ensures s.items =  $\Lambda$  and s.max_size = max_size
    function get_max_size (preserves s : stack) returns max_size : integer
      ensures max_size = s.max_size
    function get_size (preserves s : stack) returns size : integer
      ensures size = |s.items|
    procedure push (alters s : stack, consumes x : T)
      requires |s.items| < s.max_size
      ensures s.items = #s.items o #x
    procedure pop (alters s : stack)
      requires s.items /=  $\Lambda$ 
      ensures  $\exists x : T, \#s.items = s.items o x$ 
    procedure pop (alters s : stack, produces x : T)
      requires s.items /=  $\Lambda$ 
      ensures #s.items = s.items o x
    function isempty (preserves s : stack) returns empty : boolean
      ensures empty iff s =  $\Lambda$ 
    function isfull (preserves s : stack) returns full : boolean
      ensures full iff |s.items| = s.max_size
  end Stack_Template
```

Figure 2—Bounded_Stack_Template specification in RESOLVE.

Complete Encapsulation

In RESOLVE, the predefined operations for every data type are swapping, initialization, and finalization. This is true for *all* types, and it is the presence of these operations for every type which allow for complete encapsulation. All other operations must be declared explicitly by the author of the type. All parameter passing is done by swapping, rather than by value, so there is no implicit copying by parameter passing. All objects are automatically initialized when they are declared, and all are also finalized before the block is exited. Note that all of these mechanisms can be accomplished in Ada by strict adherence to guidelines.

Efficient Implementation

Since the fundamental data movement operation is swapping rather than copying, there are no copy semantics in this version of the *Bounded_Stack_Template*. Although there are no functions in this unit, note that RESOLVE functions do not imply copy semantics

for their return values. Before the assignment, the previous contents of the receiving variable are finalized, and then the value returned by the function is swapped in to the receiving variable, rather copied over the old value. Because there is no copy semantics, it is possible to implement this unit efficiently for all types *T*.

Semantic Specifications

The pre- and post-conditions are specified along with the syntax of each procedure. Functions are also prohibited from having side effects. Thus, a formal behavioral description is provided without duplicating the algorithm. The user always has an unambiguous source for clarifying his understanding of the operations of each operation, regardless of the particular implementation requested*.

Multiple Implementations

Note that the type definition of type *stack* describes the type in a mathematical notation, and that all pre- and post-conditions are in terms of this mathematical definition. This allows the type's meaning and the behavior of its operations to be described in abstract terms, without specifying the actual implementation of the data type used to achieve these semantics. This allows for multiple implementations which model the same abstract properties to be developed. RESOLVE specifically allows multiple implementations for the same specification, but all implementations must match the semantics given in that specification.

Verifiability

The presence of semantic specifications allows for the possibility of verifying implementations against them. In addition, RESOLVE has no assignment, so it is impossible to "destroy" values. RESOLVE also lacks pointers, so aliasing is not permitted. This, plus automatic initialization and finalization for all data types provides the groundwork necessary to support verification.

Ada as a Target Language

The key to using Ada as a target language for RESOLVE is that, despite the differences in the natural programming models supported by the two languages, the complete semantics of any RESOLVE module can be expressed in Ada (although some may be more painful to express than others). Moreover, all RESOLVE constructs have an efficient Ada implementation, with the exception of procedure variables. Since procedure variables may be implemented in Ada either portably or efficiently, but not both, their implementation may be inefficient. The passing of instance parameters (passing packages, such as instantiated generics, as parameters to other generic modules) may incur greater compilation time, but will still run efficiently.

* Unfortunately, as abstract data types become more complex, the formal specifications of their operations become less comprehensible. These specifications still maintain their tool-based benefits, however.

Ada does allow both the module writer and module client more flexibility in some areas than RESOLVE. Some examples of this flexibility are access types and full type visibility, neither of which is available in RESOLVE. It is the use, not just the existence, of this flexibility that prevents one from achieving the reuse goals listed above (these goals could be achieved in Ada by the strict use of enforced "programming conventions" governing the use of these language features, however the required conventions are often seen as too cumbersome). However, Ada's visibility control mechanisms are strong enough to allow the automatically generated code for a RESOLVE module to be locked inside a package, ensuring that these reuse goals are achieved inside the unit, and eliminating tampering from the client. Figure 3 shows a possible Ada implementation of the *Bounded_Stack_Template* specification.

```
with string_theory, numbers;
use string_theory, numbers;
generic
  type T is limited private;
  with procedure swap(l, r : in out T); -- these 3 routines
  with procedure initialize(x : in out T); -- are the "predefined" operations
  with procedure finalize(x : in out T); -- on the encapsulated type T.
  realization_id : string := "Standard";
package Bounded_Stack_Template is
```

```
-- Note that functions in RESOLVE translate into procedures
-- in Ada where the result is "in out". This is so the procedures can
-- finalize the variable to contain the result before placing
-- the answer into it (and also because the types are limited
-- private, so assignment isn't allowed).
```

```
type stack is limited private;
procedure swap(l, r : in out stack); -- these 3 routines are
procedure initialize(x : in out stack); -- the "predefined" operations
procedure finalize(x : in out stack); -- on the type stack.
```

```
procedure set_max_size (s : in out stack; max_size : in integer);
  -- requires max_size > 0
  -- ensures s.items = Lambda and s.max_size = max_size
function get_max_size (s : in stack) return integer;
  -- ensures max_size = s.max_size
function get_size (s : in stack) return integer;
  -- ensures size = |s.items|
procedure push (s : in out stack; x : in out T);
  -- consumes x
  -- requires |s.items| < s.max_size
  -- ensures s.items = #s.items o #x
procedure pop (s : in out stack);
  -- requires s.items /= Lambda
```

```
-- ensures ThereExists x : T, #s.items = s.items o x
procedure pop (s : in out stack; x : in out T);
-- finalizes current value of x, then
-- requires s.items /= Lambda
-- ensures #s.items = s.items o x
function isempty (s : in stack) return boolean;
-- ensures empty iff s = Lambda
function isfull (s : in stack) return boolean;
-- ensures full iff |s.items| = s.max_size

private
  type real_stack_type;
  type stack is access real_stack_type;
end Bounded_Stack_Template;
```

Figure 3—Bounded_Stack_Template specification in Ada.

In addition, clients written in Ada will be able to reuse compiled RESOLVE code as easily as other RESOLVE modules. Of course such clients will have to follow certain conventions which are required by the RESOLVE module (such as initializing and finalizing data elements) but which are unenforceable from within the RESOLVE module itself. In addition, RESOLVE specifications can be written for lower level Ada units which obey RESOLVE conventions (i. e., provide the required primitive operations on new data types and follow the required programming guidelines), and then higher level RESOLVE units can reuse this code. [Harms 89a] gives the best overview of the RESOLVE programming paradigm, its differences from more traditional approaches, and efficient implementation methods.

REFERENCES

[Gargaro 87]

Gargaro, Anthony , "Reusability Issues and Ada," *IEEE Software*, July, 1987.

[Harms 89a]

Harms, Douglas E. and Bruce W. Weide, *Types, Copying, and Swapping: Their Influences on the Design of Reusable Software Components*, Ohio State University, March 1989, OSU-CISRC-3/89-TR13.

[Harms 89b]

Harms, Douglas E. and Bruce W. Weide, *Efficient Initialization and Finalization of Data Structures: Why and How*, Ohio State University, March 1989, OSU-CISRC-3/89-TR11.

[Muralidharan 89]

Muralidharan, S. "On Inclusion of the Private Part in Ada Package Specifications," *Proceedings of the Seventh Annual National Conference on Ada Technology*, March, 1989.

[Muralidharan 88]

Muralidharan, S. and Bruce W. Weide, *On Distributing Programs Built from Reusable Software Components*, Ohio State University, November 1988, OSU-CISRC-11/88-TR36.

[Weide 86a]

Weide, Bruce W., *Design and Specification of Abstract Data Types Using OWL*, Ohio State University, January 1986, OSU-CISRC-TR-86-1.

[Weide 86b]

Weide, Bruce W., *A Catalog of OWL Conceptual Modules*, Ohio State University, January 1986, OSU-CISRC-TR-86-2.

A Model Solution for the C³I Domain

Charles Plinta

Software Engineering Institute

Introduction

This paper¹ briefly describes a specific portion of recent work performed by the Domain Specific Software Architecture (DSSA) project at the Software Engineering Institute (SEI) - the development of a model solution for message translation and validation in the C³I domain. Based on this experience and our involvement with programs in the C³I domain, future considerations are described. These considerations involve identifying potential models within a domain and making recommendations for developing and documenting model solutions that will enable the model solutions to be reused.

Background

The work was performed in the C³I domain by Charles Plinta, Kenneth Lee, and Michael Rissman, specifically in conjunction with the Granite Sentry Program. Granite Sentry is a phased hardware and software replacement of the systems in the Cheyenne Mountain complex of NORAD. The DSSA project supports the program office by attending reviews and providing advice on technical issues. In addition, the DSSA project members participate in the design discussions and working group meetings with the lead designers. As part of our involvement the DSSA project developed a model solution to perform message translation and validation (MTV). The MTV model is currently being used by Granite Sentry Phase II in its design specification and the MTV model solution will be used to implement that portion of the design. The MTV model solution is also being used by other programs developing systems in the C³I domain: Strategic Command and Control System (SCCS) and MCC/MSS.

An Overview of C³I Systems

Figure 1 shows a high level block diagram of a typical C³I system. The *Gateway* sends messages to and receives messages from all external systems. The *Gateway* is an interface between the C³I system and all other systems. Messages² are communicated between systems. The messages enter and leave the C³I system as *external representations* of the information whose formats are defined by the external systems.

The *Mission Processor* maintains a view of the world based on the views (*external representations*) provided by the other systems. This world view is kept in an *internal representation* to allow processing of the information based upon the C³I systems mission requirements. This view is available to other systems via *external representations* of the information and the user via *user representations* of the information.

¹This work is sponsored by the U.S. Department of Defense. The views and conclusions contained in this paper are solely those of the author(s) and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

²A message contains pieces of related information

The user interface (*User I/F*) provides a window into the mission processor's view of the world. It presents all or a subset of the world view, as requested by the user, in a form that is understandable to the user. The user can also add information to the *Mission Processors* view of the world. The messages enter and leave the *User I/F* as *user representations* of the information whose format is understandable to a user.

The *Journal* is a storage device used for "safe" storage of all representations of messages for recovery, analysis, and testing purposes.

Finally, Figure 2 shows a simple example of the different representations of information in a specific message.

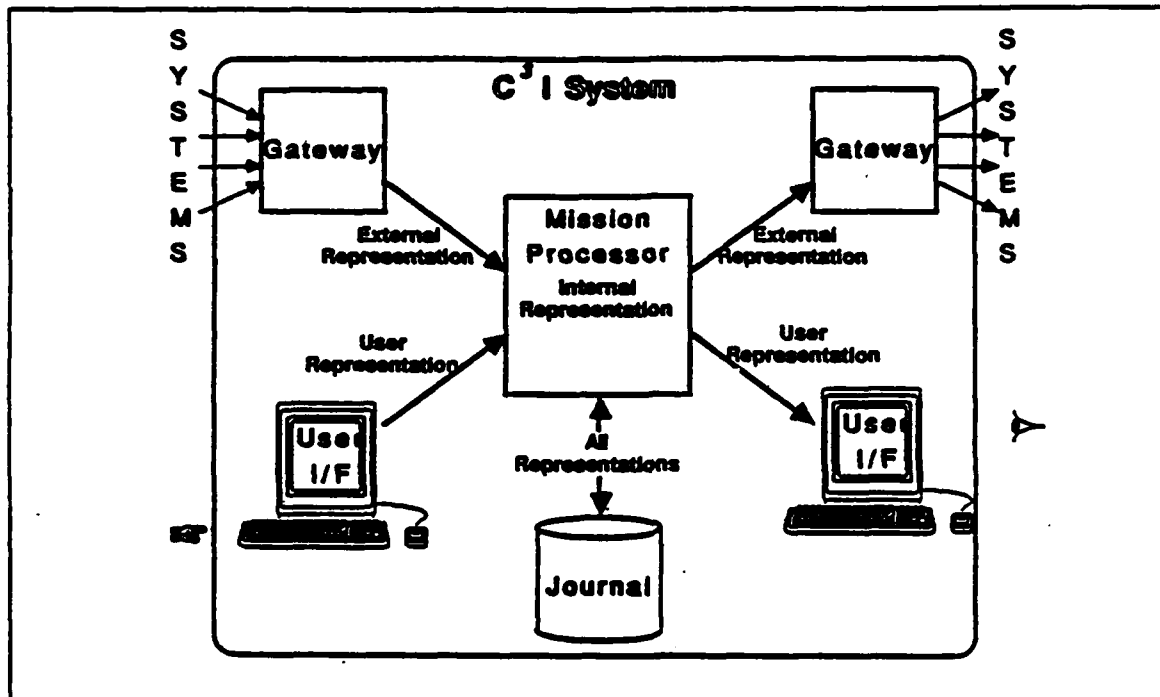


Figure 1: C³I System Block Diagram

External Representation:

"A/1810244/1<cr>"

Internal Representation:

```
Message := (Location => Peterson_AFB,
            Date      => (Julian_Day => 181,
                          Hour       => 2,
                          Minute     => 44),
            Status    => Operational);
```

User Representation

" Peterson_AFB 181 2 44 Operational"

Figure 2: Message Representations

The Problem

Based on an analysis of Granite Sentry specifically, and the C³I domain in general, we arrived at the following MTV requirements.

1. Support real-time activities:

- a. Translation and validation of external message representations to internal message representations (and vice-versa) to support mission processing.
- b. Translation and validation of all message representations to support writing to a journal.

2. Support non-real-time activities:

- a. Generation of external message representations to support simulation scripts for training purposes.
- b. Generation of all message representations to support system testing.
- c. Translation and validation of all message representations to support reading from a journal.

3. Support interactive activities:

- a. Translation and validation of external message representations to internal message representations (and vice-versa) to support manual entry of information along with presentation and correction of invalid information received.
- b. Translation and validation of user message representations to internal message representations (and vice-versa) to support manual entry of information along with presentation and correction of invalid information received.

The MTV Model Solution

This paper will not attempt to go into the details of the solution because we are limited on space³. Instead, we will present an overview of one of the two parts of the MTV model solution, the typecaster model solution.

Typecaster Model Functional Description

The typecaster model provides the capability to convert between either the user representation or universal representation⁴ of a message and the internal representation of a message. The conversion entails a real-time validation that includes syntactic analysis of the range of values possible for the elements and checking of any inter-element dependencies. If a problem is found, the conversion process is stopped, and the caller is notified. The typecaster model also supports a diagnostic, non-real-time syntactic analysis of both user representations and universal representations. A diagnostic indicator is returned that supports error detection for both user representations and universal representations of a message. Figure 3 shows a black box diagram of the typecaster model.

³An SEI technical report SEI-89-TR-12 entitled "A Model Solution for the C³I Message Translation and Validation" is forth-coming.

⁴A universal representation is an intermediate representation of the information in a message. It is the external representation with punctuation removed and all fields padded to fixed lengths. The universal representation of the message shown in figure 2 is "A18102441"

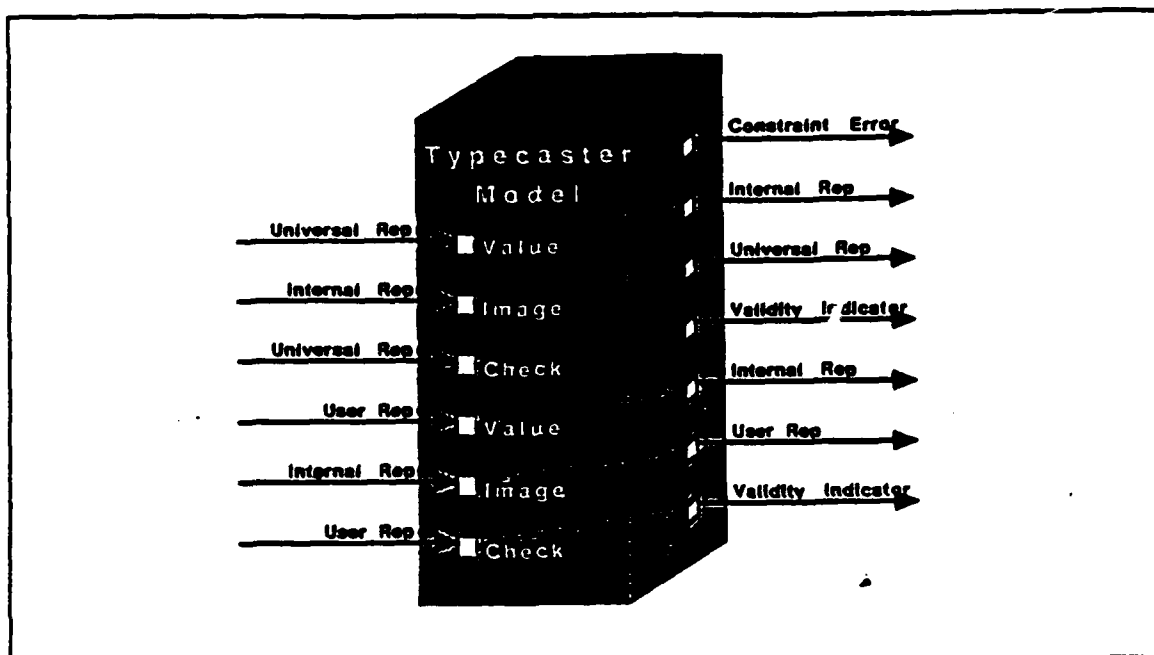


Figure 3: Typecaster Model Black Box Diagram

Typecaster Model Solution Building Blocks

The parts of the typecaster model solution fall into three categories. All the components are necessary to provide the functionality of the typecaster model described above.

1. **Discrete Typecaster Generics** - Ada generic packages that serve as the foundation of the typecaster model solution. These must be compiled into the Ada library for use by other portions of the typecaster model solution.
2. **Discrete Typecaster Templates** - Ada coding templates⁵ that are the building blocks of the typecaster model solution. The templates provide the capability to perform typecasting on Ada discrete types. Instances of these templates are layered upon the discrete typecaster generics. The templates also provide a test procedure that does exhaustive testing, based on the range of the Ada discrete type, and interactive testing.
3. **Composite Typecaster Templates** - Ada coding templates that are also building blocks of the typecaster model solution. The templates provide the capability to perform typecasting on Ada composite types. Instances of these are layered upon instances of both discrete typecaster templates and other composite typecaster templates. The templates also provide a test procedure that does canned testing, based on test cases supplied when the template is instantiated.

Typecaster Model Solution Building Plan

The following are the steps involved in applying the typecaster model solution to a set of messages that need to be translated and validated.

1. **Compile Foundation Utilities** - The utilities that form the foundation of the typecaster model solution must be compiled. These are the components in the Discrete Typecaster Generic category.

⁵A template is a file containing an incomplete Ada package specification, body and test procedure. The incomplete parts of the code are marked with placeholders. The template is instantiated by supplying information in place of the placeholders via editor substitutions.

2. **Analyze Message** - Define the internal representation description (Ada type) based on the information provided in the description of the external representation. An Ada type for each field must be defined.
3. **Instantiate Typecaster Model Solution** - Use the templates provided by the typecaster model solution to create an instance of the model solution based on the internal representation that results from the message analysis performed in the previous step.
 - a. **Identify and Build the Discrete Typecasters** - The discrete typecasters needed to translate and validate the discrete elements of a message are identified based on the Step 2. Check to see if any instances of them already exist; some may have been created for other messages. Generate the discrete typecasters that don't exist using the appropriate discrete typecaster templates. Run the generated test routines to check the discrete typecasters.
 - b. **Identify and Build Composite Typecasters** - The composite typecasters needed to group discrete and composite elements of the message are identified based on Step 2. Check to see if any of them already exist; some may have been created for other messages. Generate the composite typecasters that don't exist using the appropriate composite typecaster templates. Run the generated test routines to check the composite typecasters.
 - c. **Build the Message Typecaster** - The message typecaster is generated using the appropriate composite template, usually the record typecaster template. Run the generated test routine to check the instance of the typecaster model solution for the message.

The user of the model solution need not be concerned with the generics unless the code performance (sizing or timing) is not adequate to meet his requirements. The user need only be concerned with the templates and instantiating them as necessary to obtain the MTV capabilities required by the system under development.

Typecaster Software Architecture

Figure 4 shows the general software architecture that results when the typecaster model solution is applied. The software architecture is shown as Ada packages and the dependencies among them.

The typecaster portion of the software architecture is based upon the structure of the Ada type. When the typecaster model solution is instantiated for a particular message, the resulting architectural components are instances of the discrete typecaster templates and composite typecaster templates, one for each type used to describe the internal representation of the message. The typecaster architecture is therefore hierarchical in nature. The discrete typecasters are dependent upon the discrete typecaster generics. The composite typecasters are dependent upon both discrete typecaster instances and composite typecaster instances.

Conclusions

The DSSA project has developed a MTV model solution for a problem that recurs in the C³I domain. Granite Sentry Phase II is using the MTV model solution. The functionality provided by the MTV model solution meets their needs, and based on early timing and sizing analysis it also satisfies their performance requirements. Also, they are able to generate and test the software to translate and validate a typical message in less than two hours.

While developing the MTV model solution and participating in design reviews at Granite Sentry, we developed a process for identifying models. This process entails identifying problems that recur on a

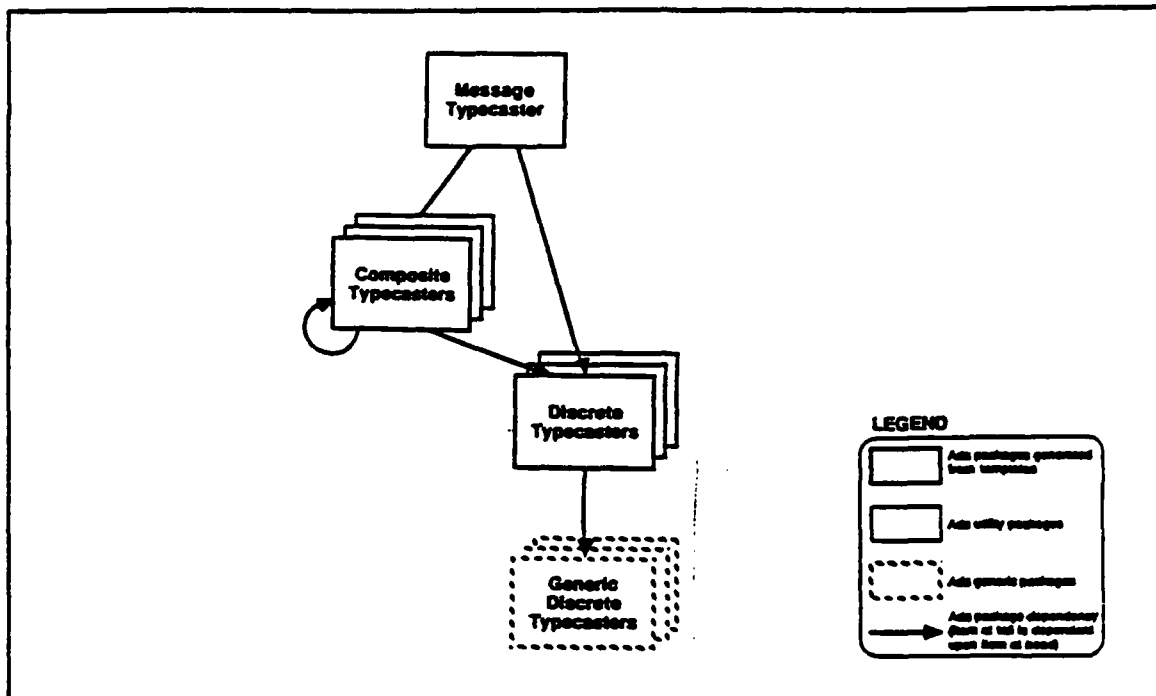


Figure 4: Typecaster Model Solution Software Architecture

project or across projects in a domain. Once identified, solutions to these problems are developed and made into models. Also, while developing the MTV model solution, we developed a way of documenting models to make them recognizable, usable, and adaptable⁶.

Based on our experiences developing, documenting, and transitioning the MTV model solution to the C³I domain, we feel that the development and use of domain specific models in the software engineering field will provide high payoffs.

Future Considerations

To achieve these payoffs, domain specific model bases must be populated and the software development process must be refined to take advantage of an existing pool of model solutions. These should occur as an evolutionary process.

First, domain experts need to identify recurring problems in their domains and develop model solutions for them. We will attempt to validate and refine our recurring problem approach for identifying targets to model by applying it in several domains.

Second, model solutions need to be developed and verified. Based on our experience with Granite Sentry, solutions should be developed and verified for a few instances of the recurring problems. The instances should also be tied together to demonstrate that the models can be integrated to meet system requirements. Verification should be based on both functionality and performance. After the solutions are verified, the

⁶Rich D'Ippolito was instrumental in helping to define how a model solution should be documented to make it reusable at both the design and implementation levels.

next step is to generalize the model solutions using code templates. The code templates help to insure that each instantiation of the model provides the functionality that is specified by the model. The templates also promote code and comment consistency. These characteristics of the template should also promote reuse.

Third, models solutions need to be documented so that they are recognizable, usable, and adaptable. We propose the following documentation outline:

1. **Problem Description** - (*everyone*) Describes the problem the model solves.
2. **Model Description** - (*designer*) Provides a functional and interface description of the model.
3. **Model Solution Overview** - (*designer and detailed designer*) Provides an overview of the model solution. Lists the parts, how to apply them, and architectural ramifications of the use of the model solution.
4. **Model Solution Application Description** - (*detailed designer and builder*) Describes how to use the model solution to solve your problem.
5. **Model Solution Detailed Description** - (*builder, maintainer and model adapter*) describes the details of the model solution implementation.
6. **Model Solution Adaptation Description** - (*designer, model adapter model adapter*) Describes how to adapt the model solution if it doesn't quite solve your problem.
7. **Open Issues** - (*everyone*) Issues of interest to everyone. These include functional limitations, performance limitations, etc.

Finally, the development process needs to be refined to encourage systems to be designed by selecting the appropriate models from the model bases, verifying designs based upon model solutions and finally, building the system using the model solutions used to verify the designs.

**A STUDENT PROBLEM TO WRITE A GENERIC
UNIT FOR A REUSABLE COMPONENT**

**BY RUTH RUDOLPH
COMPUTER SCIENCES CORP.
MOORESTOWN, NJ**

One of the assignments in an intermediate Ada programming class is to write a package to implement an Abstract Data Type (ADT). The particular ADT that is to be implemented is a set. Sets exist in Pascal as the "Set Data Type". Ada has been criticized for not including the set data type in the language definition. Therefore this problem provides an opportunity for the student to:

1. Show how easily the Ada language can be extended.
2. Create the reusable component - an ADT for sets.
3. Demonstrate the ease and degree of reusability for this generic unit.

PROBLEM

The student is given the following two part problem:

Part 1

A common application of sets is in the realm of numbers. We frequently refer to the set of integers, the set of prime numbers, the set of natural numbers, and so forth.

The student is asked to write a program that computes the following for numbers between 1 and 100 inclusive:

- . The set of numbers divisible by 2, 3, or 5
- . The set of numbers divisible by 2 or 3, but not by 5
- . The set of numbers divisible by 3 and by 5
- . The set of numbers not divisible by 3

To support the solution the student is provided with two library units:

1. The specification of the ADT for sets
2. A function to compute multiples, to generate sets of all numbers that are multiples of some integer and are in the universe provided by the set package.

The student must write the package body and write a driver to use the library units, i.e. the set package and the function.

Part 2

After completing Part 1, the student is asked To rework the problem by making the set package a generic and including a generic instantiation for the reusable unit in the driver. The student discovers that placing the instantiation in the driver denies access of the ADT package to the function. The function requires two definitions to be in scope before it can be compiled. First the base type which describes the set universe (the actual type which will be used for the instantiation) must exist. Second, the insert procedure which is advertised in the set package specification must exist.

There are at least four possible solutions:

1. Embed everything in the driver. The most obvious solution is to declare the actual type and instantiate the generic in the driver procedure. The function also must be embedded in that procedure. A slight change has to be made in initializing the constant sets. They can no longer be initialized in the declarative part because the Ada rules require basic declarations to precede later declarations. Since the function is no longer a library unit, but a later declaration in the driver, the constant set object cannot use the function to obtain their

initial values. This may be unimportant but it is no less a consequence of using the generic.

The student no doubt will select the above solution. But suppose this was a more complex example and the embedded function was very useful as a library unit. Is it possible to retain it as one?

2. Put the actual type, the instantiation and the function in a package. Although this is a possible solution it does not seem to accomplish very much other than to simplify the driver. Again, the problem is that the function must be preceded by the actual type definition and the instantiation.
3. Put the type and generic instantiation in a package and embed the function in the driver. Similar to solution two, this doesn't offer any real advantage over the first solution, but at this point in the investigation one's curiosity has been aroused.
4. Put the type and generic instantiation in a library package and the function in the library. If the goal was to maintain the function as a separate library unit, this is the only workable solution. It does, however, require an extra package which may become burdensome in a large library.

Although this may appear to be a trivial exercise the lessons learned from this endeavor are of great value:

1. Identifying a generic candidate may appear obvious but may present problems that are not intuitively identifiable.
2. Implementing a program as a generic may be relatively easy but may not take into consideration certain factors that are relevant.
3. Using the generic easily and without rewriting or changing the organization of the original program may not be possible.

An Informal Experiment in Reuse

Roger Van Scoy and Charles Plinta
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Introduction

This paper¹ focuses on the practical impact of reusable software in system design. This paper is the result of work performed in creating a software artifact and, as such, the insights presented are based solely on that experience. The purpose of this paper is to use that experience to provide insight on ways to facilitate the use of software components.

This exercise in reuse resulted from work done by the Software Engineering Institute on the development of a prototype Real-Time Monitor (RTM) for Ada applications in support of the Ada Simulator Validation Program (ASVP)². An RTM is, in its simplest form, a tool that can read and write data (e.g., variables) in an executing application; it is essentially a primitive, remote debugger. Our task was to build a tool familiar to the ASVP contractors (from their flight simulator experience) that executed in conjunction with an Ada application. The concepts needed to build an RTM were not new, but interfacing one to an Ada application in a distributed environment was. Since the RTM was built for use in conjunction with real-time applications, it was designed to execute in a CPU's spare time and minimally perturb the essential timing of the application.³

The RTM task was a modest effort, as illustrated by Table 1⁴. In addition to being a practical and useful artifact (for the ASVP contractors), the large percentage of existing software components used in building the prototype make it an excellent vehicle for discussing some of the issues related to using software components. We start by distinguishing between the two categories of reuse:

1. Design reuse: The reuse of concepts or software components providing functionality that satisfies design element specifications. The Virtual Terminal, Command Line Interpreter, and Forms Management subsystems fall into this category.
2. Implementation reuse: The reuse of software components providing functionality that aids in implementing a portion of a design element specification. The Binary Tree or Linked List abstract data types fall into this category.

This paper will focus on some of the practical issues related to design reuse.

¹This work is sponsored by the U.S. Department of Defense. The views and conclusions contained in this paper are solely those of the author(s) and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

²ASVP was a research and development contract sponsored by the program office for Training Systems at Wright Patterson Air Force Base. The contract called for the redesign and implementation of two existing flight simulators in Ada using sound software engineering techniques. The contract was awarded to two contractors with the objective of learning lessons about developing flight simulators in Ada.

³See [3] for an overview of the project and its results.

⁴One person worked full time for three months on the implementation.

<u>Lines</u>	<u>Type of Code</u>
1,643	Command Line Interpreter subsystem
2,421	Virtual Terminal subsystem
2,291	Forms Management subsystem
1,593	Abstract Data Types (Binary Tree, Linked List, etc.)
7,948	Total reused lines
1,054	New code
9,002	Total lines

Table 1: Ada Statement Count for RTM

Reuse in Design

Looking back on our design work, we feel that knowledge of reusable software components that implement well understood, high-level concepts should be used during the design process. To realize this in practice however, places a number burdens on the designer.

First, the requirements of the system must be stated abstractly enough to allow design reuse to occur. It is easy for implementation biases to inadvertently creep into the requirements or design, resulting in an over-specified system. Systems must be specified abstractly and concisely. The distinction is that a specification must indicate *what* is to be done, rather than *how* it is to be done.

For example, one component in our top-level design was a command line interpreter. Figure 1 illustrates some differences between an abstract specification and an over-specification of a command line interpreter:

<u>Specification Part</u>	<u>Abstract Specification</u>	<u>Over-Specification</u>
1. What it needs	a command	a string(1:80)
2. What it provides	command field interpretation	strings, integer #s, real #s
3. What it does	parsing of command line	LALR1 parser
4. What resources used	< x% of available memory	no dynamic memory allocation
5. What could happen	errors in command	raise command_error exception

Figure 1: Command Line Interpreter Specification

While the over-specification is useful if one is building a command line interpreter, it has constrained the design element to the point where only a hand-crafted component will meet the specification. The abstract specification on the other hand, is general enough to allow many possible command line interpreters to meet the specification. Thus, designing with the intent to reuse components forces the designer to carefully consider the ramifications of each requirement or design element; concentrating on what (design) rather than how (implementation).

Second, reuse implies a certain flexibility on the part of the designer. Once a set of potential components has been identified, one must choose the "best" one. Ideally, a perfect match is desired—one where the component meets the requirements and is compatible with the design. This rarely happens, posing two problems:

1. What criteria does one employ to measure potential components for closeness of fit?
2. Based on a close fit and a decision to use a particular component, what is involved to fit the component into the system?
 - a. Does one alter the requirements and/or design to fit the new component into the system?
 - b. Or does one modify the component to fit the requirements and/or design?

Clearly, the first step in access the suitability of a component is a "technical best fit" analysis. The designer must evaluate the potential components with respect to the requirements and the design. This analysis involves creating a set of prioritized design criteria that must be satisfied by any acceptable component and grading each potential component according to these criteria.

Assuming that one or more components meet the "technical best fit" criteria, the decision to actually use the component requires a "management best fit" analysis. The criteria we used in performing the "management best fit" analysis were:

Requirements	Must the requirements change to accommodate the component? If so, what are the system ramifications and the cost of the change?
Design	Must the design change to accommodate the component? If so, what is cost of the change and how does it impact on work in progress, other components, and the design documentation?
Component	Must the component itself be modified? If so, how much will it cost to modify the component? A critical factor here is, whether the authors of the component will make the needed component modifications. If not, then there is a need to analyze the documentation and implementation to determine the effort required by the new engineers to pick up the component and modify it.
Schedule	How does the decision to reuse or not to reuse affect the overall project schedule?

For instance, one difficult issue we faced was extending the functionality of the Forms Management subsystem into an area for which it wasn't explicitly designed. We opted to modify the component rather than the design, as described below.

The Forms Management subsystem had four parts to it:

1. A part that builds form templates,
2. A part that allows the user to fill in a form,
3. A part that displays filled-in forms, and
4. A part that allows an application program to extract data from the form.

These parts were implemented to operate as independent programs in a non-real-time environment. Our task was to integrate them into a system that allowed forms to be defined interactively and filled in dynamically in real-time. The task of integrating the parts was non-trivial and could not be done by looking at the available documentation and code. Rather, it required that we prototype the subsystem and study the problem experimentally before deciding to proceed with the component. In the end, the Forms Management subsystem was successfully extended. But at what cost? In this case, the savings were substantial, since this represented about 25% of the total code in system. But, had the integration effort failed, the time spent on the mini-prototyping effort would have been wasted and the needed component constructed by hand.

Clearly, we don't feel it is a simple matter to pick out a component and "slap" it in place. Neither the cost nor the risk of reuse are negligible. The decision to use an existing component or build a new one must be carefully evaluated. The cost advantages of adapting an existing component are potentially tremendous; likewise the cost of failure becomes potentially exorbitant. Again, the burden here is on the designer to formulate and apply the acceptance criteria for each component.

This brings us to the actual components themselves. All the components we used were obtained from the Ada Software Repository⁵ via a manual search process (an automated search mechanism would have been an enormous help). We found a wide variation in component documentation. This leads us to suggest a minimum set of information necessary for specifying components. Two levels of information are needed:

1. A high-level description of the functionality provided by the component.
2. A detailed description of the component (along the lines of IEEE 1016 [2]).

The high-level description of the functionality provided by the component must include the familiar concepts provided by the component. The concepts embodied by a component must be well known and thoroughly understood for a component to be recognized as reusable. This implies the need for domain-specific component libraries, where the background of potential users gives them a common vocabulary with the component implementors. This information will be needed to locate candidate components.

The detailed description of the component should include the following information:

1. What it needs. Specification of the information that the component needs to perform its job or change its state.
2. What it provides. Specification of the information that the component makes available after performing its job or changing its state.
3. What it does (not to be confused with how it does it). Specification of the functionality provided by the component, a description of its job. This should include a description of the concepts provided by (embodied in) the component.
4. Performance documentation (especially in the real-time world). This needs to include items like: timing and memory needs, dynamic versus static allocation schemes, automatic garbage collection versus manual garbage collected, etc. (See [1] for example).
5. Rationale-type documentation. It is not sufficient to simply explain how something is done in code; this can be gleaned from the code itself. What is needed is why a specific action is performed or why the component is implemented in a specific manner. This type of information is of great value, since it gives the user a glimpse into the mind of the component's designer.
6. Test software and documentation. Perhaps the primary obstacle to component reuse is the adequacy of the testing the component has undergone. Without documented evidence of the quality of the component, this work must be repeated with every use of the component. This process is made more difficult because the software is often unfamiliar. The inclusion of test mechanisms and expected results along with the component is critical.

This information is essential to performing the technical and management best fit analysis discussed above. The burden of documenting the components lies with the component designers and implementors.

⁵For more information call (703) 685-1477 or write Ada Software Repository, 3D139 - The Pentagon, Washington, DC 20302-3081.

Conclusion

Clearly, reuse of software components can be important in building software systems, but it places requirements on both the component users and implementors. These requirements indicate a need for additional work in the several areas:

- Requirements generation: Approaches to requirement and design specification that allow for and incorporate reuse need to be developed and practiced.
- Analysis techniques: Techniques and methods for deciding when a component is acceptable or when a component must be built.
- Component libraries: There is a need for domain-specific libraries containing *quality*, tested components. Also, any library of significant size needs a taxonomy and a database to aid the potential user in locating components of interest.
- Documentation: An information content standard needs to be developed and required of all components in a library.

Acknowledgement

Our thanks to the other Dissemination of Ada Software Engineering Technology (DASET) Project team members, Michael Rissman, Richard D'Ippolito, Kenneth Lee, and Timothy Coddington, who contributed to the Project on which this paper is based.

References

- [1] Booch, Grady.
Software Components with Ada.
The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.
- [2] *IEEE Recommended Practice for Software Design Descriptions.*
The Institute of Electrical and Electronics Engineers, Inc., 1987.
Std 1016-1987.
- [3] Van Scoy, R.
Prototype Real-Time Monitor: Executive Summary.
Technical Report CMU/SEI-TR-87-35, Software Engineering Institute, November, 1987.

Locating Resources For Reuse-Based Development¹

Sholom (Sanford) Cohen
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
NET: sgc@sei.cmu.edu

Abstract

Locating resources in a collection of reusable software requires integration of management and retrieval methods. The maintainer of the collection will need different operations and modes of access from an application developer using the collection. Users of the collection will want access to it at different phases of the life-cycle. Finally, users will have expertise in different application domains, but may desire access to the same information.

This paper briefly examines methods for managing and locating reusable resources. These methods include both attribute- and facet-based information retrieval. The paper also introduces a new method to support search and retrieval that captures information specific to an end-user application (both requirements and design) and matches that information to resources in the library. Integration of these approaches is essential for successful reuse-based software development.

1. Introduction

Successful application of reusable software assumes that a user can find and use software from a library of reusable resources. A catalog maintenance scheme to control the resources and data about them and an information retrieval scheme to obtain the resources and data for the user are essential. Current techniques for implementing these schemes center around traditional data base models and library classification methods. These techniques are appropriate for code components intended for a range of applications, where decisions about potential use occur during software design and implementation.

These traditional data base and classification methods limit the utility of a library of domain-specific reusable

software. These methods are directed at the detailed design and coding stages of software development. However, for applications in the areas of avionics or command and control, the decision process for determining the appropriateness of a reusable resource, even code components, should be concurrent with the software requirements and early prototyping phases. Without knowledge of the reusable resources, requirements specification can restrict the use of specific resources. In fact, certain key systems decisions, such as the choice of specific hardware or algorithms, may totally preclude the use of a reusable resource during the later stages of software development. Software specification decisions made in light of existing software resources, as hardware design decisions are currently made in light of existing hardware components, will result in an increase in software reuse.

Current mechanisms for cataloging and retrieving code components are aimed at software engineers, at a point in the development when systems design and software requirements are complete. The mechanisms assume that the user has firm software requirements, and possibly design, and is searching for code components to be used in implementation. Furthermore, most existing requirements for reuse are built upon established computer science models, e.g., abstract data types, tools, etc. These assumptions are not appropriate for the real-time embedded systems applications engineer who must make decisions in such areas as choice of hardware devices, searching and tracking techniques, and Kalman filter methods. The applications engineer requires strategies for finding and retrieving not only code components but also other reusable resources based on the application, not on the components themselves, and must employ these strategies early in the development. This paper will describe methods for retrieving reusable

¹Sponsored by the U. S. Department of Defense

software that also address the needs of application engineers.

The next section of this paper summarizes the problems in managing a collection of reusable resources. The section describes why proper management of resources must account for different sources, various classes, and non-uniformity in packaging of reusable software. The next two sections of the paper contrast and show examples of existing methods of locating resources in a library. Examples of the search techniques will demonstrate their utility in a variety of settings. The concluding section will also emphasize the importance of integrating these methods for management and retrieval.

2. Reusable Resource Management

The management of a large collection of reusable software resources poses a serious storage and control problem. Management of the resources must take into account control of the information to initially store the resources and to support changes in the information due to maintenance. This type of collection management differs from the traditional view associated with library support, due to the dynamic nature of a reusable software collection and to the fact that the collection is primarily electronic. However, studying contemporary trends in library science can offer support to the management of reusable software.

Software collections will be built from reusable resources from a variety of sources. The collection may come entirely from a single project or set of related projects within a single organization, or the source may be at the corporate level, crossing project domains. The collection may also include resources from external sources, either government or commercial.

The complexities of managing the collection increase when the sources are corporate wide or external. Furthermore, the variety of classes of reusable software resources (code components, documentation, functional or performance test software, code generators, architectures, requirements, etc.) leads to a storage and control nightmare. Resources may fall into different categories for each class of resources, such as source code files (components), graphics (documentation and performance tests), or text (documentation). The storage methods must account for each category and the fact that a single resource may have elements from multiple categories.

In addition, the same category of resource may come packaged in numerous ways. For example, source code may come in single files (Booch WIZARD, Ada Software Repository), multiple files (CAMP), or multiple directories (GRACE Components). The manager of the collection must decide whether to attempt uniformity in storage, leading to configuration control difficulties when updates appear, or maintaining the original structure, making use of the collection difficult.

The contemporary library, or resource center, has dealt with many of these problems. Collection building, for example, takes into account the variety of sources for materials: traditional publishers, government publications, technical reports from industry or academia. Library science has also considered the storage problems for electronic media, with research in information retrieval looking at CD-ROM and hypertext. The collection and storage problems, while significant, are not the main topic of this paper, however. Management of the collection must also include control of the methods to search for and retrieve information about the resources. The remainder of this paper describes techniques for supporting this second function.

3. Attribute-based Retrieval

The management of reusable software resources requires the development of appropriate attributes and methods for storing and retrieving them. In traditional cataloguing, referred to as descriptive cataloguing in library science, the resources of a collection are described through their attributes. For reusable resources, an attribute might be the name of a part, the function of a subsystem, or other information about the resource. For searching a catalog of reusable resources, one specifies a specific attribute or a combination of attributes and values for those attributes. The searcher obtains resources that have those values and is provided with information about the resources. This descriptive information is also in the form of attributes of the resource.

Successful use of this retrieval mechanism requires an understanding of several key factors:

1. The meaning of the attributes
2. The range of values a specific attribute may take
3. The structure of the collection of resources
4. The ability to combine or add attributes

5. The relationship of the attributes to requirements for an application

This level of understanding suggests that the user is highly familiar with the collection of components he is dealing with. Primarily, this method of searching for reusable resources is aimed at a software engineer, with knowledge about the specific resources. The method is most appropriate at design time. The manager of the collection will also use this approach.

4. Semantic-Based Retrieval

Semantic information about a resource tells what that resource can do in a given context. The user of this search technique does not require an understanding of the contents of the collection because the search method is based on what the resource does in an application, rather than on how it fits into a collection of other resources.

The method is useful at any stage of development. Because the search is not built around the resources themselves but around what they can do, a user may be a systems engineer unfamiliar with the specifics of the software functionality captured by the resources and unfamiliar with software terminology used to describe the attributes of the resource.

The faceted approach is an example of retrieving resources through information about what they do. A facet is defined as a class of terms that characterize one way of looking at a resource. A complete faceted classification of a collection of reusable resources consists of several different facets, each with its own list of terms.

The SEI Application of Reusable Software Components Project developed a faceted approach to classifying reusable resources. The project used the Asset Library System (ALS), developed by GTE Labs to support its faceted classification and retrieval. The project established a reuse collection consisting of the CAMP components, the GRACE Components from EVB, and the Booch WIZARD components. These components have been arranged into three collections:

1. Missile operations
2. Kalman filter and mathematical operations
3. Abstract data types and utilities

For each collection there is a separate set of facets and terms, specific to the components in that collection. For example, in the missile operations collections, there are

the following facets:

1. Subsystem
2. Role
3. Purpose
4. Input
5. Output
6. Method

For each component in this collection, there will be one set of terms for each facet. For the component *Compute_east_velocity* the terms are:

Facet	Term
Subsystem	Navigation
Role	Operator
Purpose	Wander-azimuth
Input	Nominal east & north velocities Sine & cosine of wander angle
Output	True east velocity
Method	Sine-cosine

This method has the advantage of being very flexible and easy to extend. If new components come into a collection, terms may be added to existing facets, if necessary, to fully classify them. If a completely new collection of resources is added, then a new set of facets is created, specifically for that collection. The classification will then be done with the new facets and terms.

The method is most effective for a library of closely related, low granularity, single function resources. For this type of collection, the faceted method provides excellent discrimination between similar components. If a collection is diversified or more complex software is introduced, the faceted method becomes less effective. For diversified collections, additional facets are required. Complex software requires numerous terms in each facet for complete classification, making the terms difficult to use. These factors weigh against this method for use at the requirements stage, when diversity of functionality and top level issues are the concerns in assessing the application of reuse.

5. Application-Based Retrieval

The application-based method uses information about complete systems to identify appropriate resources for reuse on an application. The method was developed by the Common Ada Missile Packages (CAMP) program for the parts identification subsystem of the Ada Missile

Parts Engineering Expert (AMPEE) system. Most of the information needed to implement this method comes from data captured during the domain analysis that led to the implementation of the reusable resources themselves. This approach proceeds in parallel with reuse development, rather than following the development as in the attribute and faceted approaches.

There are two methods for performing application based retrieval. A system approach focuses on application features or capabilities and retrieves resources based on the specific features of a given application. The second method works from a generic architecture or model of software for the domain and matches resources to the version of that architecture as customized for a specific application.

Because the application-based approach focuses on a high-level view of the system under development, the technique is appropriate for the systems designer as well as the software designer and can be used early in the development life-cycle for identifying reusable resources. The next two subsections describe these two approaches in terms of the information the user must provide, the search strategy, and the data returned to the user.

5.1. System Approach

The interface for the system approach is at the systems level. Under the approach, the user provides information about the system under development. This may include general requirements, or specifics about subsystems. The search strategy will judge the requirements for consistency and consider alternative means of utilizing available resources in support of the user's requirements. The approach will return to the user the catalog attributes of the resources that support his requirements. If the requirements cannot be met directly from the collection, the user will be offered suggestions for use of the resources for partial implementation of his application. In addition the approach must report on inconsistencies in the user's requirements.

The system approach provides the following features:

INTERFACE	Queries user about system requirements
PROCESS	Validates requirements for consistency Considers available resources and possible alternatives in partial support of alternatives
PRODUCT	Attributes of available resources supporting system requirements

For each resource identified, the system approach can again query the user, refining his requirements with regard to that resource. The user is not asked to make specific software design decisions, only to characterize his requirements. The system will again return resources matching these requirements.

5.2. Software Architecture Approach

The software architecture approach provides the user with a graphic depiction of architectures for a class of applications. The interface allows the user to tailor the general architecture to the specific architecture of the system under development. This customization involves selection of subsystems and functions to support specific software requirements.

The processing step must assess each tailoring step and link that customization to the available reusable resources. As the user navigates the generic architecture, he will eventually arrive at specific resources which may be Ada components or other reusable software.

The architecture method provides the following features:

INTERFACE	Generic architectures of typical applications User "builds" custom architecture from generic
PROCESS	Considers possible alternatives at each level of architecture Links custom architecture to available resources
PRODUCT	Attributes of available resources supporting custom-built architecture

The next two subsections illustrate the application of these two approaches to the selection of CAMP software.

5.3. AMPEE Missile Application Exploration

The AMPEE system contains a wide range of functionality and resources to support software retrieval.

- menu interface - captures application data
- graphics interface - depicts generic architectures, supports user selections
- knowledge base - stores inheritance network to build architectures
- rules base - relates system requirements to architectures/parts, supports consistency

checks

- catalog data base - controls part attributes and searches on attributes

The system approach in AMPEE utilizes these features in interacting with the user. The user enters data about his system through menus. AMPEE uses this data in performing consistency checks through the knowledge and rules base and in selecting appropriate CAMP software from the catalog data base to implement the features. The parts list is then passed to a catalog function storing attributes of specific resources.

The following structure explains the manner in which AMPEE implements the system approach.

INTERFACE	Menus query user about system requirements
PROCESS	Knowledge base supports consideration of possible alternatives Rule base to validate for consistency
PRODUCT	Parts list passed to catalog function to retrieve attributes of available resources supporting user requirements

The AMPEE tool returns a parts list that leads to a continuing dialogue with the user to refine his selection of components.

5.4. AMPEE Missile Model Walkthrough

The Missile Model approach allows a user to build a simplified version of his application from a standard architecture. It provides a graph of architectures for the application area that showing reusable software in support of the implementation of high level features. Nodes on the graph will be the subsystems and functions supported by specific CAMP software. The user traverses the graph, selecting nodes to customize the architecture of his own application. The AMPEE system accepts the identification of a node by supplying the next level of the architecture. At the lowest level, the nodes on the graph are individual resources of reusable software. The features support by this method are:

INTERFACE	Graphics depict a generic architecture User walks through the architecture
PROCESS	Knowledge base provides data at each level Links custom architecture to available CAMP software
PRODUCT	Displays attributes of available resources

The user begins by identifying several of the subsystems of his specific application from the generic missile model. At successive levels of the architecture the user can select more specific resources. The output to the user is, once again, the attributes of the reusable software stored in the catalog. The user may query these as he proceeds through the graph or may accumulate them to the end.

6. Summary

Management of a collection of reusable software resources will pose challenges to those wishing to build a large reuse library. The problems of building the collection, describing it, and classifying it, must account for the variety of resources that are available for reuse and the differing requirements of users of the collection. The methods for managing the collection, both in storing and controlling information, must, therefore be flexible and extensible. They must also consider not only managing the resources, but also supporting the goals of reuse-based software development.

Methods for retrieving information about the collection must also address these same goals. Users will have skills from different domains and will access the library at various points in the development life-cycle. Those unfamiliar with the structure of the collection must still be given access to the resources, and those who have knowledge of the contents of the collection must be provided with powerful tools to capitalize on their expertise. These goals can be accomplished by providing methods that address both what the resource *is* as well as what the resource *does*. Integration of retrieval methods to support the attribute, semantic, and application-based approaches will provide this required level of support for users of a reusable software collection.

MANAGING LARGE REPOSITORIES FOR REUSE

April 10, 1989

Prepared by:

Beverly J. Kitaoka

Science Applications International Corporation
Innovative Technology Group
Science Technology & Software Operation
Ada Software Division
311 Park Place Boulevard, Suite 360
Clearwater, FL 34619



Science Applications International Corporation (SAIC) is in the process of building and operating three significant Ada repositories for reuse: Software Technology for Adaptable Reliable Systems (STARS), Air Force Logistics Command (AFLC), and SAIC Corporate. While each repository has distinctive needs, the majority of the needs are common. The STARS repository serves as a testbed for repository and reuse technology. Distinctive needs include a means for sharing program-related information such as meeting minutes, presentations, Contract Data Requirement Lists (CDRLs), and peer reviews. The AFLC Ada software repository has been established to provide a means of sharing information among the software development and maintenance personnel at the Air Logistics Commands. With the wide variety of domain-specific software developed at SAIC, the company has decided to institute a reuse program to enhance the quality of the domain software and encourage cross-domain reuse.

The primary source for repository technology has been, and will continue to be, derived from the STARS program. This paper will describe the experiences and challenges of creating and operating useful repositories with this technology.

The major repository issues have been divided into four categories: content acquisition/update, technical information services, facilities, and operational support. Content acquisition/update includes classification, evaluation, and cataloging; technical information services include supply, reuse, logistics, and forums; facilities issues include platform, equipment, and communications; and operational support issues include access, usage monitoring, tool installation/customization, machine operation, and configuration management.

Content Acquisition/Update

We decided to establish these repositories by collecting all the Ada software we could get our hands on. While the objective is to provide a repository of certifiably reliable components, we determined that first we would need to learn how to make certifiable components (through creation or adaptation). We felt that the process of evaluating a large collection of software would teach us a lot about the characteristics of software, the classification of software, and the adaptation of software for reuse. This initial repository state has been designated a "depository (junkyard)" by the IBM Houston STARS team. The Houston team has defined a spectrum of repository classes which includes: depository, filtered, organized, managed, and certified. If one considers a repository to be a collection of software work products and their supporting information, it is possible to consider a repository of repositories in which collections can be in the various stages mentioned above.

With this in mind, we created depositories of the following software collections for the STARS, AFLC, and SAIC repositories: ALS, ASR/SIMTEL20, CAMP, IDA Ada/DIANA Front End, IDA Ada/SQL Binding, NOSC/WIS, SDME, and UNITREP. In addition to these collections, the STARS repository contains STARS Foundation and STARS Prime software, the SAIC repository contains SAIC proprietary software, and the AFLC repository will contain software specific

to the Air Force. We are in the process of obtaining AFATDS, U.S. Army/CECOM, McDonnell Douglas and Ada SAGE, U.S. Marine Corps and determining how to make the STARS software available to organizations not affiliated with the STARS program.

The initial IBM STARS repository used the VAX/VMS directory structure to organize the code by program source, using subdirectories that paralleled those used by the distributing agency. Simple directory keyword and source text searching capabilities were implemented using VMS utilities. This approach provided minimal organization to the repository while the contents were analyzed for key characteristics and reuse potential.

As our next step in organizing the repository contents, we chose a hierarchical, relational data model implemented on Oracle, a widely used commercial Database Management System. Oracle runs on several platforms, including the PC, which will be useful in implementing the shadow repositories and project libraries. We chose the hierarchical approach because we felt it would be easy to implement. Our next goal is to implement a faceted data model. The faceted approach offers higher extensibility, flexibility, precision, and succinctness over the hierarchical model. Since this approach has been successfully implemented on small, domain-specific libraries, we have decided to enlist the efforts of experienced individuals to assist with the transition of a large collection of diverse software from a hierarchical to faceted model. We will retain the source directory and hierarchical approaches as alternatives to the faceted model. Implementation of the hierarchical model will provide comparison data for the faceted model on this type of repository.

To "organize" this large collection of software, we have constructed guidelines for software engineers to evaluate the contents of the repository. These guidelines include a copy of the baseline data model; descriptions/explanations of the database tables and potentially obscure items; a master template (electronic) for entering the characteristics of the product under evaluation into the database; a description of reuse categories; user manuals for tools which provide further information about the product, such as lines of code; reuse definitions and guidelines; sample product reviews; type lists; and ownership categories. The goal of the evaluation effort is to learn as much about a given product as possible for classification and reuse potential. This information will be useful in transitioning from a hierarchical to faceted data model and providing information for repository users. We intend to use these guidelines to process new repository acquisitions and updates.

The evaluations will help us "filter" the repository by providing information on redundancy, outdated versions, incomplete or nonworking software, and form and content consistency.

We have created a tool which will generate a catalog entry for each product in the database using the data entered via the template and generated by the evaluation support tools. This catalog tool will allow us to generate an up-to-date copy of the repository catalog whenever new products are acquired.

Part of the classification effort for the AFLC repository required domain analysis for two selected domains: onboard flight programs and flight simulation software. We determined that the classification scheme, or taxonomy, was dependent on the domain analysis and, therefore, a product of the domain analysis. This determination was reinforced by discussions with Ruben Prieto-Diaz and literature searches. An incomplete domain analysis hindered the process of locating potential reuse candidates for classification under the domain, requiring further domain investigation and reclassification of domain-specific components.

Objectives of the next STARS increment include guidelines for creating "managed" and "certified" repositories. We will apply these guidelines to the STARS repository as they become available.

Technical Information Services

Technical information services support the repository users. These services may currently be invoked from screen menus and database queries. We are exploring other interfaces such as Natural Language Interfaces, Hypertext, and the use of Graphics as alternatives to the menu and structured command approach.

The process of a user supplying a work product to the repository is similar to the content acquisition/update process described above, only the supplier will provide the classification and database information instead of the repository personnel. Guidelines and templates for entering this data will be supplied to the user. The repository personnel will then evaluate the software and load the database. This process will be tested in the next STARS increment as STARS work products are created.

In the first STARS increment, tools were selected from the repository to add capabilities to the STARS software engineering environment under development. Interface standards, "Virtual Interfaces," were created for the environment. The objective was to remove the existing tool interfaces, replace them with the Virtual Interfaces, replace any code with existing reusable components where possible, and create new reusable components from code contained in the tool. This turned out to be a more productive exercise than we first realized. The reuse of the Virtual Interfaces and components produced a large amount of working code in a small amount of time. We faced performance problems in some areas by reusing components which were too general for the application; we had to deal with components which were not at the right level of abstraction, carrying unnecessary baggage into the application; we had to deal with the necessity of variations for different application platforms; and we had to deal with some reuse management problems.

Reuse management problems included the construction of new code because the software engineer was unaware of the existence of a particular reusable component. This was primarily a problem with the initial directory classification structure. The immediate fix for this problem was to emphasize reuse in the design reviews - the problem was discovered during

the code reviews. Another problem discovered involved adaptive reuse of a component. In modifying a component for use in a schedule-critical application, the software engineers did not pay attention to making the adaptations reusable and returning them to the repository as reusable component variations. The impact was discovered when the same changes were needed for another application, and the component had to be extracted from its now application-specific environment and modified again for general reuse. The fix was to stress reusability as well as reuse.

Repository tools which we have created from salvaged repository software include: browser, annotation editor, pretty printer, standards checker, profiler, compile orderer, cocomo, SGML processor, catalog generator, and evaluation generator.

We are in the process of incorporating our reuse guidelines with our software development and quality assurance standards and procedures.

Logistics, the process of getting the right product to the right place in the proper form, will be part of the next STARS increment work.

To increase communications among repository users, forums have been created using the VAX/VMS NOTES product. This capability allows users to enter information on a given topic and solicit responses. It has proven useful in locating solutions to problems raised by the STARS team members.

Facilities

To provide a useful repository, the facilities must support the required repository capabilities which include evaluating, certifying, and storing the contents. The current IBM STARS repository contains 1.7 million lines of Ada statements requiring 214 megabytes of memory. The total size of the repository is 367 megabytes.

The platform selected for these repositories is a VAX 3600 running the VMS operating system. This platform was selected for the SAIC and AFLC repositories primarily because the equipment was already in use by each organization. It was selected for the STARS repository because of the wide usage of the VAX platform among the STARS participants and the modular expansion of resources available by clustering the VAX computers. The VAX/VMS platform also provides sufficient Ada support so compilation can be part of the evaluation process.

The significant number of software and hardware vendor offerings for the VAX/VMS platform provides a basis for rapid-prototyping repository capabilities. Commercial off-the-shelf products can be used on a trial or purchase basis to determine the requirements for repository operation and usage capabilities. These requirements can then be used for make/buy decisions based on current and future considerations. Future considerations may impose performance requirements across distributed, heterogeneous platforms, for example.

Equipment for the repository includes 32 megabytes of physical memory, approximately 2.4 gigabytes of disk storage (four DEC RA82 drives), a cartridge tape drive (DEC TK70), and a multi-density tape drive (DEC TU81). Of the four physical drives, one is allocated as the system disk, one is allocated as user disk space, and the other two are bound as one logical disk volume (ADA\$REPOSITORY) where the repository contents reside. An optical disk capability will soon be added to increase the variety of physical media distribution. Access to the IBM STARS repository is primarily via dial-up modems into the SAIC terminal server with automatic connection to the repository VAX. There are currently eighteen telephone lines organized in a rotary group. Sixteen of the eighteen lines are connected to 2400-bps modems equipped with MNP Level 5 error correction and compression. The last two lines are connected to 9600-bps modems equipped with MNP Level 6 error correction and compression. The nineteenth line serves a dual purpose, as a trouble line and for modem testing, and is not in the rotary group. A network bridge connecting the SAIC Clearwater Local Area Network (LAN) and the IBM Gaithersburg LAN has been installed to improve communications between the two IBM STARS teams. Efforts are being made to obtain a dedicated ARPANet node.

Ultimate repository platforms will probably include distributed heterogeneous computers - allowing optimal performance for a given capability. UNIX platforms, including the new IBM platforms supporting AIX and the BiIN computer, are being studied for this purpose.

Operational Support

Operational support capabilities include providing user access, monitoring usage, installing and customizing tools, operating the machines, and performing configuration management.

At this time, only STARS affiliates can access the STARS repository, and only SAIC employees can access the SAIC Corporate repository. We are currently working on methods to resolve this access restriction. Access to the AFLC repository is in the process of being established. Usage monitoring will provide information on who uses the repository and how they use it. This information will give us insight to problems with repository support and assist in the definition of new requirements. The absence of hits for database queries will provide demand information which will be used to identify the need for specific reusable components.

Configuration Management guidelines have been developed using the VAX/VMS CMS product. These guidelines will be improved in the next STARS increment.

SAIC will continue the work of enhancing these repositories while exploring variations of the repository, such as shadow repositories and project libraries. The STARS program has been a valuable means for developing the technology applied to these repositories.

POSITION PAPER
REUSE IN PRACTICE

Aerospace Distributed Software Library

William Novak

GE Resident Affiliate
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
NET: wen@sei.cmu.edu

1. Background & Motivation

1.1. Introduction

Virtually every programmer maintains a personal library of reusable code and uses it regularly in developing new software. Often some of the most effective software developers are those who have at their disposal a large personal collection of previously developed software that they can apply to new problems. The difficulty is that no other programmer knows enough about the contents of that personal library to be able to use it as well. As a result, time is wasted developing independently something that could be done better and faster by working together.

1.2. Task Description

In response to this situation GE Aerospace has developed an automated system for the classification, storage, search, and retrieval of reusable software. The Aerospace Distributed Software Library, or ADSL (formerly known as RARS), currently provides automatic GE DECnet retrieval of over one thousand modules written in Ada, FORTRAN, C, and other languages which have been catalogued from GE, public domain, and commercial sources. The ADSL offers such search techniques as taxonomy-based (using a hierarchical classification structure), keyword, and text search, with various search constraints available to limit searches based on different properties of the software.

The ADSL has been implemented using the INGRES relational database with Standard Query

Language (SQL) and runs under the VAX/VMS operating system. The library units themselves exist as VAX files that are distributed on various nodes across the GE DECnet network, and the system automatically retrieves the units selected by the user with a background job at the end of the ADSL session. This arrangement keeps the amount of disk space required for storing the Library software at any particular node to a minimum.

1.3. Purpose

The purpose of the ADSL development is to create a distributed library of reusable software across all of GE Aerospace using this system to catalogue and retrieve the library units. The ADSL supports not only several different search techniques, but an extensive Library Maintenance application that allows all aspects of the system to be maintained, so that the different ADSL sites may be managed independently.

The current contents of the ADSL have been taken from many sources, but represent only the smallest beginnings of what the Library will contain. In 1987 hundreds of Ada components were taken from the public domain SIMTEL-20 Ada Software Repository to establish an initial set of Library units for testing. In 1988 two additional reusable software libraries have been added: 1) the Booch components library, 500 different data structure packages written in Ada by Grady Booch, and 2) the Ada generic components library developed at Corporate Research and Development in Schenectady. Software of any type may be catalogued in the system— there are

no restrictions on type, application, implementation language, or even quality—but all of these attributes are recorded in the Library catalogue entry, so that if a user is interested in software that is, for example, written in Ada and that has passed acceptance-level testing, then only the software units meeting those constraints will be seen.

2. Aerospace Distributed Software Library

2.1. System Services

The system currently offers four primary options to the user:

1. Search for and retrieval of software units from the library
2. Review of current library metrics data
3. Viewing of recent library news items and user comments
4. Maintenance of the library catalogue data

2.2. Search

The *Search* option is the one of most interest to the general user. Users may search for software using the hierarchical software classification taxonomy, keyword searches, or text searches. Multiple taxonomies for different software development domains exist side by side, and software units may be catalogued in several different taxonomies and/or taxonomy classes if needed. The user may browse up and down through software taxonomy classes that become increasingly specific, or specify a class of interest and move directly to that class. An extensible *thesaurus* will accept common abbreviations and translate them into keywords or full class names.

To limit the search, users may specify certain search constraints to retrieve (for example) only software that runs on a specific operating system. All searches may be constrained by the computer the software runs on, the operating system it runs under, the implementation language it is written in, the amount of testing it has passed, and the date it was catalogued into the library. Another type of

search constraint which is available allows for distinguishing between reusable components which are identical in their basic function, such as a set of many "stack" packages which are implemented in different ways. They may differ in the ways they handle concurrency or memory management, and the system allows the search to be constrained according to these attributes.

All units that are found using any search method will be displayed to the user and upon request, the code and associated documentation will be retrieved into the user's current default directory.

2.3. Metrics

The *Metrics* option is used to view various statistics and metrics that have been collected on overall library usage. Metrics may be viewed by library unit, site, or individual library user. Metrics are gathered on the number of library users, the number of library units, the number of retrievals by sites or users and the average frequency of retrieval of any given unit, plus various other metrics.

2.4. News and Information

The *News and Information* facility allows the user to view information on recent acquisitions to and enhancements of the library. It also allows users to view and edit lengthy comments about the performance of individual library units.

2.5. Library Support

Finally, a separate facility, *Library Maintenance*, exists for the cataloguing of new Library units and the maintenance of various other Library information. This facility is used by the Library Administrator and can define classification taxonomies, delete library units, and modify unit descriptive information.

3. Conclusion

The fact is that there are still no commercially available systems that fully support software reuse and are suited to the distributed GE environment. The existence of a sophisticated software retrieval system within GE Aerospace is critical to

answering the increasing demands for improved productivity. The current ADSL system provides a practical, efficient, and sophisticated software retrieval system that is being put into place at GE Aerospace sites now, and will be able to grow along with GE's requirements for software reuse.

Position Paper
for
Reuse in Practice
Workshop

Pittsburgh, PA

11-13 July 1989

Constance Palmer
McDonnell Douglas Missile Systems Company
Dept. E434, Mail Code 0922232
P.O. Box 516
St. Louis, Missouri 63166
(314) 925-7930

REUSE IN PRACTICE

There has been talk of software reuse for many years, yet in reality the full potential has yet to be reached. In the past several years, as software costs have sky-rocketed, there has been an increased focus on reuse as a means of controlling or reducing those costs, and as a means of producing a higher quality product.

Some domains have been more successful than others at applying reuse technology. For example, the Japanese have been very successful in some banking and telecommunications areas, and companies in the U.S. have reported success with other business applications. One area that has been particularly problematic is real-time embedded (RTE) applications — developers of these applications are particularly constrained in terms of both space and timing requirements, and generally have very stringent reliability requirements. There has been (and still is) a great deal of skepticism among this group of developers about the viability of software reuse in their applications.

The Common Ada Missile Packages (CAMP) program (being performed by the McDonnell Douglas Missile Systems Company, and sponsored by the Air Force Armament Laboratory at Eglin Air Force Base) is aimed at addressing the issues of software reuse in an RTE domain. The author has been involved in the CAMP program since it began in 1984, and thus, the perspective on reuse presented here is in terms of reuse of Ada in RTE applications. Phase 1 of the CAMP program (referred to as CAMP-1) was a feasibility study. The primary objectives were to determine if sufficient commonality existed within the missile operational flight software domain to warrant the development of reusable software parts, and if commonality was found, to identify and specify the parts. The feasibility of automating aspects of parts engineering was also explored, and the requirements and top-level design for a parts composition system were developed. During Phase 2 (CAMP-2), the parts were coded and tested, as was the parts composition system (PCS). Both the parts and the PCS were used in the so-called "11th Missile Application". Additionally, a set of armaments (armament electronics) benchmarks was developed. The 11th Missile Application was a demonstration program that involved the use of both the reusable Ada parts and the parts composition system in the development of a realistic application. Phase 3 (CAMP-3) is currently underway. The main tasks are maintenance and enhancement of the CAMP Ada parts, re-engineering the prototype PCS catalog function in Ada, and development of a manual on "Developing and Using Ada Parts in Real-Time Embedded Applications".

1. TERMINOLOGY

As in most developing technology areas, there is no clear consensus on the definition of terms in the software reuse area, thus it is important for authors and speakers to define what they mean. It may as yet be too early to standardize on a complete vocabulary, but one's perspective should be clarified for the audience. Over time, as reuse becomes more widespread, a common set of definitions will emerge from the reuse community.

It is difficult to come up with one set of definitions because domain and scope impact the meaning/interpretation of different terms. For instance, efficiency requirements within a payroll system and efficiency requirements within missile operational software may be orders of magnitude different, yet developers of both may discuss "efficiency" and assume that their meaning is clear. There is, of course, some overlap in definitions, but it is best that they be clarified to prevent misunderstanding.

2. REPOSITORIES

A repository is an essential aspect of a viable software reuse methodology. It is a central facility for storing information about available software parts, and may, in fact, contain the parts themselves. A significant amount of work has been done in this area, and a number of alternative structures and parts attribute sets have been proposed and developed. Because reuse is and will be practiced at many different organizational levels (e.g., project, company, corporate, industry, domain, etc.), no one structure will suffice for all instantiations of a parts repository.

(1) Scope

One of the most important preliminary parameters that needs to be defined before a meaningful discussion of repositories can ensue is that of *scope*. Scope will impact the features of the repository that are affected by user diversity, which, of course, increases as the scope is broadened. Scope will affect features such as the type of user interface that will be needed (user interfaces are an important consideration in repository development, particularly as software engineers become more accustomed to multi-window graphical interfaces in their other software engineering tools), security requirements and access control, accessibility, robustness requirements, and domain covered by the repository.

Scope affects support needs as well. For example, a repository that is intended for use by a project or company, may need just a librarian to verify that coding and documentation standards have been met before a part is added to the repository. Other cursory checks could include checking for test documentation and results. A large-scale, community-wide, or even

corporate-wide, library might require a significant staff to not only check code and documentation for compliance with standards, but also perform independent testing, assess the *value* of parts, assist users in the use of parts (throughout the lifecycle), develop new parts, provide training, etc..

As the scope broadens, the need for and demands on a repository support staff grow. It is envisioned that such a staff could be invaluable in the technology and cultural transition from custom code development to widespread software reuse. They could provide the training and support needed to alleviate risk to projects that are considering a parts-based approach to software development.

Scope impacts repositories in another way as well. As the scope broadens, the issue of who will pay for the development of new parts, maintenance and enhancement of existing parts, and user support may become significant. If the repository is for DoD mission critical application developers, should the government fund it, or should the contractors contribute to its support? Similarly, if the repository is corporate-wide, should the corporation pay for it, or should the constituent companies fund it as long as they are benefiting from it?

Under the CAMP program, we developed a parts composition system that had a parts catalog as its cornerstone. During CAMP-1, we investigated the requirements and issues associated with a large-scale catalog system, but, the prototype CAMP catalog was scoped primarily for project or company use (likewise with the CAMP-3 re-engineered Ada-based catalog). It was not so large as to make it impractical to have multiple copies within a corporation (although there are configuration control problems that can arise from distributing copies of the catalog versus allowing distributed access to the catalog).

(2) Classification

Several issues related to classification arise in the discussion of repositories. For instance,

- **What entity is to be cataloged?** In the prototype CAMP-2 catalog, we had separate catalog entries for Ada specs and bodies, as well as for structuring packages, bundles, etc.. As a result, the CAMP-2 catalog had over 1100 entries, although there were only 454 CAMP Ada parts developed. This classification scheme was problematic for the potential reuser — there were just too many entities. The user could not easily find what he was looking for. This directly contradicts the purpose of a catalog which is to (among other things) lower the cost of reuse by facilitating the acquisition of information about available parts. Others have raised the issue of whether entities such as documentation, design, or test code should be cataloged separately from the software parts themselves.
- **How are entities categorized?** During the CAMP program we were looking at a specific domain — operational missile flight software— thus, we developed a functional decomposition of the software (see Figure 1). The advantage of this categorization scheme was that it was easy to implement, although it suffers from the disadvantage that is not very flexible or extensible.

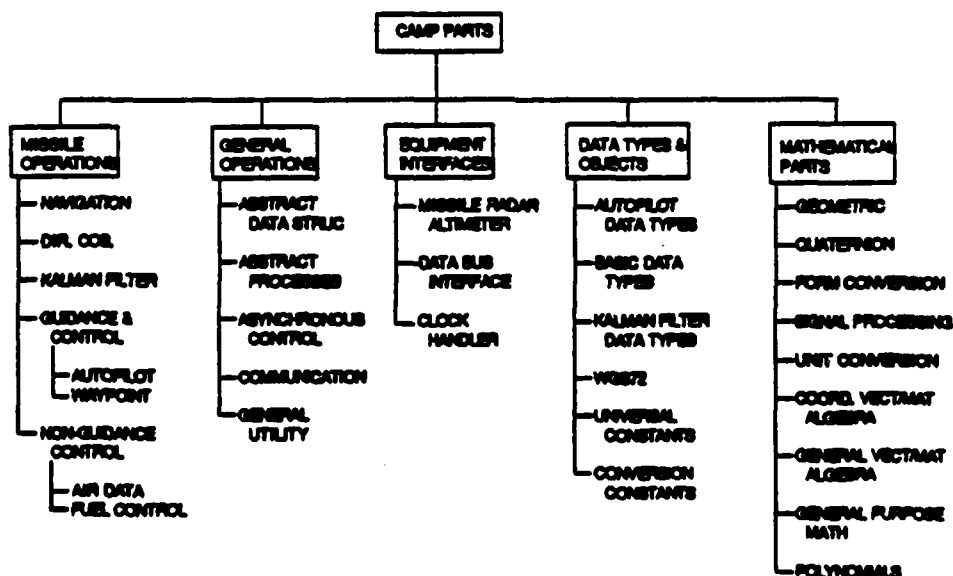


Figure 1: CAMP Parts Taxonomy

Currently, the CAMP parts catalog is being re-engineered to make it more widely usable. Although the catalog we deliver will be set up for the CAMP parts, containing the CAMP parts taxonomy, etc., a site will be able to establish a different taxonomy when they install the catalog in their environment. The taxonomy will also be modifiable, so that for example, if a site starts out with only the CAMP parts, they may want the CAMP parts taxonomy, but if they acquire another set of parts

for a different domain, they will want a taxonomy for those parts as well. The new CAMP catalog will be able to handle this situation and accommodate both taxonomies.

(3) Domain

Domain specificity depends on the breadth of definition of the domain and the number of parts. If the domain of a catalog is too narrow, users may have to access multiple catalogs in order to obtain all of the part information that they need in order to develop their application. It is difficult to establish clear cut boundaries between some domains — some domains have significant overlap (e.g., amonics and avionics). It is better to provide a structuring mechanism within the catalog than to segregate the domains externally (i.e., by putting them in separate catalogs).

3. MANAGEMENT ISSUES

There are a number of management issues that need to be explored when implementing a software reuse program within an organization. A few of them are discussed here.

(1) Management Support

Management support for software reuse is critical to its success. Organizations need to realize two things.

- They can develop a competitive advantage by developing and using reusable software in their product lines (given that commonality exists and can be developed in a reasonably cost-effective manner).
- The industry as a whole can produce higher quality products if the industry as a whole develops and uses software parts.

Once this realization comes, it is not clear that the DoD will need to provide incentives to the industry to practice reuse — the organizations will see that reuse is in their best interests. In the interim, it may be necessary for the DoD to provide incentives because of the increased cost of developing reusable software and the increased risk of incorporating it into new applications. The risk will diminish over time as the technology and methodologies emerge and mature.

The conversion to a parts-based approach to software development will require cultural changes as well as technology development. Software engineers are not taught to reuse software, they are taught to re-invent it. As Jean Sammet pointed out in Reference [4], *"Ever since the second square root routine was written, the programming field has lost adequate control of reusability."*

The effort required to develop high quality software needs to be viewed as an investment rather than as an afterthought in the overall system development. Once the investment is made, the development costs can be amortized by reusing the software in additional applications. These are attitudes that are not currently prevalent.

Management needs to have confidence that when they are asked to sign up to software reuse in their projects, that they have a reasonable chance of success. Once they are convinced of the value of software reuse, they need to be willing to make the investment in training their software engineers in both the development and use of reusable software. They need to be willing to develop a support group that can assist projects in the transition to parts-based application development. Projects are constrained by their schedules and budgets and cannot be expected to fully embrace a technology that is still somewhat risky because of its early stage of development. Management needs to be willing to share the risk with these projects.

(2) Test-bed Programs

Confidence in software reuse technology and methodologies can only be gained from success stories. Paper studies and "toy" applications will not suffice for the project people who have hard schedules and requirements constraints. Realistic test programs or shadow projects can provide the type of test environment for software reuse that can work out many of the issues or problems associated with it, and demonstrate that it is viable, cost-effective, and ultimately beneficial. The DoD can facilitate the move to software reuse by funding these types of demonstration projects.

This was one of the benefits of the CAMP-2 11th Missile application. The 11th Missile was so-called because it was not one of the original ten missile software systems that was examined during the CAMP domain analysis. The goal of this effort was to validate the concept of parts use in real-time embedded applications. The 11th Missile application entailed the redevelopment of missile guidance and navigation subsystems in Ada, using the CAMP parts. This effort was based on an existing application that was implemented in Jovial. It required the development of a DoD-Std-2167 SRS and other documentation, as well as software development and hardware-in-the-loop (HIL) testing. It was targeted at a 1750A processor. The existence of the HIL test environment made this an attractive application to parallel.

The development effort demonstrated that Ada was rich enough to allow implementation of virtually all of the required functionality (the application required only 21 lines of assembler code to accommodate existing system idiosyncrasies) and also demonstrated that the CAMP parts could be incorporated into a new RTE development effort. The 11th Missile development effort also highlighted the risks associated with the application of new methodologies (i.e., software reuse in RTE applications). Although the 1750A-targeted Ada compiler was validated, it was unable to correctly compile many of the CAMP generics (many of which are quite complex in structure). In fact, several compilers were tried. The problems encountered and the results obtained during the 11th Missile application development effort are detailed in the CAMP-2 final technical report (see Reference [2]).

If this had been a full-scale engineering development effort, the problems encountered would have been disastrous. Many of the generics had to be *manually instantiated*. The tasking overhead was prohibitive. The compiler was not able to generate object code that was efficient enough for the navigation subsystem to run in real time. Numerous lesser bugs were also encountered with the compiler.

The upside of this effort was that this work and the close interaction of the development team with the compiler vendor, spurred compiler improvements that will benefit all users of this compiler. This highlights the importance of establishing test-bed programs that can act as pathfinders and facilitate maturation of the required technology and methodologies.

(3) Return on Investment

It is too early to determine exactly when the payoff will occur in parts reuse. Although there will be gains during development — greater gains will occur as support environments mature and the cost of reuse decreases — we must also look to the maintenance phase for payoff. There we should see increased reliability in the products because of the use of extensively tested software parts. Although there is some data available in commercial business-type applications, there is not yet sufficient data available for RTE applications to determine the extent and timing of the payoff for software parts reuse.

We need to explore ways to maximize reuse and reduce the cost of reuse. This can be accomplished through training and through the development and deployment of tools that support a parts-based approach to software development. Some of these issues have been explored during the CAMP program, including prototype development of a set of integrated facilities to support software development with parts (see References [1, 3]).

4. DOMAIN ANALYSIS

Domain analysis is the analysis of a representative set of applications from a given domain with the intent of determining the common objects, operations, and structures (if they exist). Domain analysis requires first that the domain be defined. It is not always a simple process to define and bound the domain that will be considered during a domain analysis — domains often overlap. Domain analysis is tedious and time-consuming, requiring the examination of existing software documentation and source code, as well as further work with the original developers if they are still available. It provides an investigative challenge. The existence of commonality within a domain cannot be taken for granted. The main outputs from a domain analysis are (1) the identification of common objects, operations, and structures, and (2) the domain model.

Domain analyses provide the foundation upon which to build a software parts development effort, but the development of techniques and methods for performing them is still in the early stages. CAMP-1 began with a domain analysis that involved the missile operational flight software from a set of ten missiles. From this analysis, we identified approximately 250 common parts and developed a taxonomy with which to categorize those parts. We assumed (and were proved correct) that we would identify additional parts once we actually began development of the common parts. Our final part count at the end of CAMP-2 was 454.

There are as yet no widely accepted or established techniques for performing a domain analysis, but a number of issues have been identified. One factor that is critical is the selection of an adequate domain representation set upon which to base the analysis. Practical constraints prevent the examination of all applications within a domain, thus, it is important that the sample set include applications that are truly representative of the domain as it has been defined.

5. IMPLEMENTATION ISSUES

Because the CAMP domain was missile operational flight software, we were very concerned with efficiency of the parts that we developed. We had to balance our goals of efficiency, usability, and flexibility when implementing the parts. Although we developed the code to be as efficient as possible, we recognized that we might not meet the needs of all users, but felt that the end user still wins by reusing code that requires only slight "tweaking" to meet his requirements. In many cases the code will meet the user's requirements without any modification. Reuse is not an all or nothing proposition.

We also provided parts with alternative data structure representations for users with different efficiency requirements. For example, we provided a number of alternative matrix representations. If a user was relatively unconcerned with efficiency, he could use the package that provided full-storage matrix representation, but if he were concerned with space utilization he might want to use an alternative, special-purpose representation such as a coordinate matrix (if he knew he was working with coordinates). We also provided trigonometric functions with various degrees of speed and accuracy to accommodate different project needs.

Optimizing compilers are essential to the production of object code that will meet the efficiency requirements of real-time embedded applications. The CAMP 11th Missile development team provided significant feedback to the compiler developer to spur maturation of sufficiently optimized compilers.

Compilers for use with reusable software will need to be able to remove unused code within packages. Many times a package will contain more than just the routines of interest to the application developer, and these must be eliminated in order to not unduly burden the user.

The use of software from an external source need not imply that there is a security risk, or a risk of introducing a virus to one's operating environment. This is not generally a concern when using commercial software. Application developers need to be able to have the same level of confidence in reusable software suppliers as they have in their commercial vendors. This is one reason why it is important that software parts be "validated" prior to making them available in a catalog or repository. Parts repositories should provide a means for logging user feedback about individual parts; these comments can then be used by subsequent users in their evaluation of parts for their projects, and contribute to the confidence level a user can have in the correct operation of the parts.

6. ISSUES

There are many issues associated with software reuse that have not been addressed in this paper. A few of them are enumerated below.

- At what level should reuse take place (i.e., what should be reused)? Code? Design? Requirements? Ada specifications are often equated to top-level design and bodies equated to detailed design, so with this point of view, designs do get reused in Ada applications.
- What is the best architecture for use in the development of reusable software parts?
- What type of training is needed to give software engineers the skills to both develop and use reusable software?
- What time horizon should we be looking at for the technology needed to support software reuse? Are we looking at what can be accomplished now and in the near-term or what can be accomplished in ten years?
- What will the impact of reuse be of data rights and other legal issues?

7. ABOUT THE AUTHOR

The author has been at McDonnell Douglas for over 5 years, and has been involved in exploring the potential of software reuse since the inception of CAMP in September, 1984. The author was responsible for the parts composition system feasibility study during CAMP-1, and then for requirements, design, and much of the implementation of the prototype parts composition system during CAMP-2. She is currently the program manager for CAMP-3, and is exploring many software reuse issues while developing (as part of CAMP-3) a manual on parts development and use in real-time applications. The author has an M.S. in Computer Science from Washington University in St. Louis, and a B.A. in Mathematics from George Washington University in Washington, D.C..

References

1. McNicholl, D.G., C. Palmer, et al. Common Ada Missile Packages (CAMP), Volume II: Software Parts Composition Study Results. Tech. Rept. AFATL-TR-85-93, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, May, 1986. (Must be acquired from DTIC using access number B102655. Distribution limited to DoD and DoD contractors only.).
2. McNicholl, D.G., C. Palmer, J. Mason, et al. Common Ada Missile Packages - Phase 2 (CAMP-2), Volume II: 11th Missile Demonstration. Tech. Rept. AFATL-TR-88-62, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, November, 1988. (Distribution limited to DoD and DoD contractors only.).
3. McNicholl, D.G., S. Cohen, C. Palmer, et al. Common Ada Missile Packages - Phase 2 (CAMP-2), Volume I: CAMP Parts and Parts Composition System. Tech. Rept. AFATL-TR-88-62, Air Force Armament Laboratory, Air Force Systems Command, United States Air Force, Eglin, Air Force Base, Florida, 32542, November, 1988. (Distribution limited to DoD and DoD contractors only.).
4. Sammet, Jean E. "Why Ada is Not Just Another Programming Language". *Communications of the ACM* Vol. 29, No. 8 (August 1986).

The Reusability Library Framework - Working Toward An Organon

James Solderitsch

*Unisys Defense Systems
Paoli Research Center
PO Box 517
Paoli, PA 19031-0517*

**Internet: jj@prc.unisys.com
UUCP: {sdcrcf,bpa,psuvax1}!burdvax!jj
Phone: 215-648-7376**

Introduction

This paper describes work-in-progress that began under the STARS Foundations program with a contract administered by the Naval Research Laboratory (contract number N00014-88-C-2052). Current plans call for work to be continued on the Reusability Library Framework itself and on using the framework to support the construction of library systems at various levels. In addition, Unisys plans on exploring other applications areas which can benefit from the use of knowledge-based techniques implemented in systems designed and engineered from an Ada perspective.

This work has been reported on at a number of previous conferences [Solderitsch88] [Wallnau88] [Solderitsch89]. This paper will present a summary of this material and then outline plans for future development. In doing so, the paper will address some of the key issues affecting reuse technology, and the RLF approach to some of these issues. A workshop atmosphere will enable others to evaluate and critique the RLF work in a manner that is not possible during actual conferences and through private correspondence. In this way, the RLF technology can both influence, and be influenced by, developments that are underway across the spectrum of reuse-oriented projects and investigations.

RLF Approach

Ada development efforts during the 1980's have succeeded in producing an increasingly large collection of Ada components. Individual collections range from the general purpose (Ada Simtel repository, EVB-Grace commercial parts) to more narrow, application-specific collections (CAMP parts [CAMP85]). In any case, effective ways and means of collection management are required in order to take advantage of them. At the very least, support must be provided for retrieval, insertion and qualification of components in the context of a supportive library or repository management system.

Various classification schemes have been proposed [Prieto-Dias87a] but fixed classification schemes can be unnecessarily limiting. An approach that permits a library organisation to evolve along with the components being kept in the library is better able to support software collections as they change both in size and maturity. Moreover an adaptable library organisation is better able to serve the needs of focused application domains. Important semantic attributes of software artifacts are often dependent on the domains to which they belong. As such, library support software must be semantically tailorable to represent and use such attributes.

The RLF technological approach is founded on the premise given in the preceding paragraph. Broad objectives of the RLF project include:

- develop knowledge-based interfaces to repository (object) management systems;
- investigate the mapping between application domain and reuse technology (part selection, part composition, part generation);
- go beyond supporting retrieval of static parts to include program generation, system/software configuration, system/software testing and even system/software design and requirements analysis;
- support the basic integration of reuse technologies (knowledge-based and generation techniques); and,
- perform some applied research in domain analysis.

On a smaller, near-term, scale, the RLF aims to:

- develop foundations technology essential for building "intelligent libraries" for reusable software components within specific application domains; and,
- develop as part of the RLF reusable, stand-alone components supporting integration of knowledge-based techniques into other Ada applications (beyond the library domain).

The approach taken by Unisys seeks to overcome some of the weaknesses apparent in other classification-based reuse support systems. One important aspect is the accessibility of the classification scheme itself and the relative ease by which the classification data base can be tailored and extended. Moreover, the RLF provides the user with guidance on the use of the classification system so that the user is not forced to become an expert in the classification scheme to use it effectively.

Advantages of Domain-Specificity

A cornerstone to the RLF is its reliance on a domain-specific point-of-view. In the context of the RLF, a *domain* is comprised of a set of existing and anticipated software applications that provide a common function or similar capability. Domains can be further sub-divided into horizontal and vertical domains. A horizontal domain is one (e.g. common data structure definitions and operations) whose contents intersect with vertical domains oriented around a company's line-of-business (LOB) or specialized applications area.

History has shown that many of the past success stories for reuse have come within certain well-defined domains (e.g. mathematics routines). Unisys believes that the impact and successful application of a reuse-based approach to software design and

production will be greatest for (vertical) domain-specific libraries. For example, a greater proportion of a typical application can be built from parts withdrawn from such a library. There is also a higher expectation that systems built from such parts will have a closer functional fit and be more efficient. The capability exists for reusable sub-systems to be created via part selection and configuration.

There are real costs in establishing such a library and not every domain is mature and stable enough to support such an intensive reuse-based approach. Domain analysis [Prieto-Dias87b] to support such libraries can be hard, and is certainly expensive and time-consuming. However, domain analysis is a fundamental prerequisite for a reuse environment to support the extended life-cycle of an application domain. Such support is analogous to the way that some software engineering environments support the traditional waterfall life-cycle. The goal of domain analysis is to provide fundamental support for the organized growth and development of software applications for the domain, both from the consuming side and producing side of the software equation.

RLF Overview

Figure 1 illustrates the basic architecture of the RLF at the end of the STARS Foundations contract phase of the project. The final product of this project was a prototype librarian application covering the domain of Ada benchmark programs.

All components of this system were developed from an Ada perspective using basic principles of data abstraction, information hiding and strong typing. Abstract data types were produced after analyzing the structure of proven Knowledge Representation Systems (KRSs), first by focusing on the operations provided by these systems, and only later considering possible internal representations of knowledge held within the system. No attempt was made to naively import features native to AI programming language paradigms such as pattern matching or theorem proving.

AdaKNET is a semantic network system useful for capturing static information describing the basic state of some enterprise or subject area. For example, our two initial uses of AdaKNET were to capture some basic Ada semantics regarding Ada compilation unit structure and portions of the Ada type lattice for use by Gadfly, an Ada unit test plan generator; and, to represent some basic relationships among Ada benchmark programs for use in an Ada benchmark program library system. An important part of our work concerns how to combine the representational power of AdaKNET with other systems, including other KRSs.

AdaTAU is a rule base system that can be used as a stand-alone system or in conjunction with other knowledge representation systems such as AdaKNET. Rules collected into rule bases are used to infer new facts from a collection of initial facts. New knowledge is added to a system employing the facilities of AdaTAU so that AdaTAU is acting like an expert system that enhances the capabilities of the original system. When used together with a system like AdaKNET, AdaTAU becomes part of a hybrid KRS where the role of AdaTAU is to facilitate the capture and use of dynamic information that is normally outside the realm of the other cooperating KRS. For example, the benchmark librarian rules are used to advise librarian patrons of operational information regarding benchmark components that are not easily discernible within the benchmark taxonomy provided through AdaKNET.

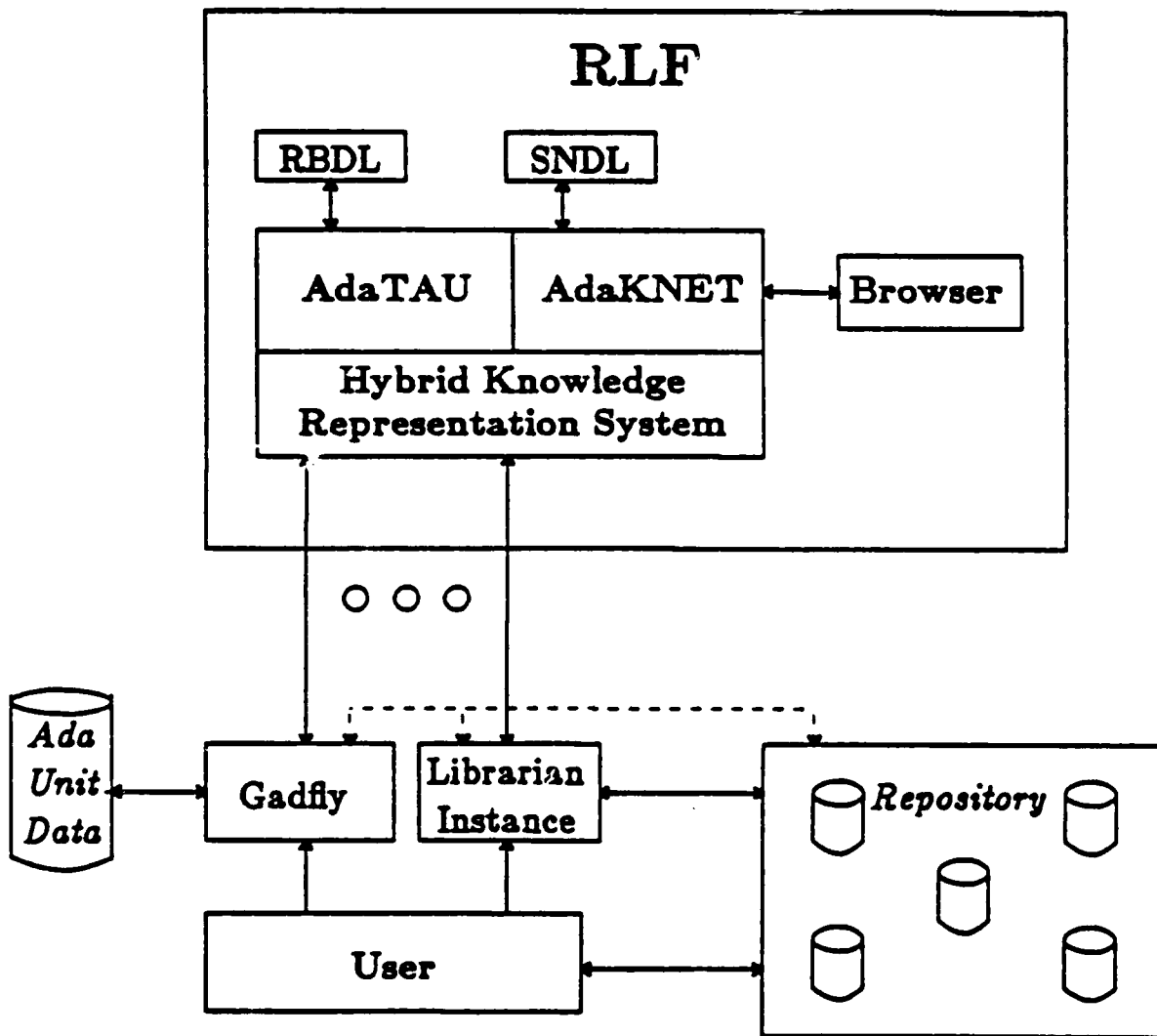


Figure 1. RLF Architecture

A careful separation of the content of knowledge bases from their basic organisation and available operations is provided through the use of two specification languages developed explicitly for the RLF [Solderitich89]. RBDL (Rule Base Definition Language) and SNDL (Semantic Network Definition Language) are used to specify rule and fact base descriptions for AdaTAU and semantic network descriptions for AdaKNET respectively. Individual knowledge base definitions are translated automatically to an Ada compilation unit that, when executed, produces a machine

readable version of the original specifications. The design and implementation of these specification languages was accomplished through the use of SSAGS - Syntax and Semantics Analysis and Generation System [Payton82]. SSAGS itself is an Ada-based tool developed at Unisys that is especially appropriate for the specification of small, application-specific languages (ASLs) and their translators.

The end user typically works directly with an application built on top of AdaKNET, AdaTAU or a hybrid of both of them. In addition, an application makes use of its own data structures. For example, in using the Gadfly application, knowledge about an Ada unit under test is assembled and stored within a hybrid knowledge base. From this knowledge gained by examining the Ada unit directly and as a result of a dialogue conducted with the user, suggested test case plans are generated for the user. For the librarian user, a repository of Ada modules is available for direct examination. Alternatively, the user can browse, or be "expertly" guided through, an information web that captures essential information about the contents of the repository. A library patron offering a new component for the repository can be guided to the right insertion point and, using an integrated form of Gadfly, be advised of necessary quality control measures to be taken before the component can be officially installed.

Knowledge Representation and Librarians

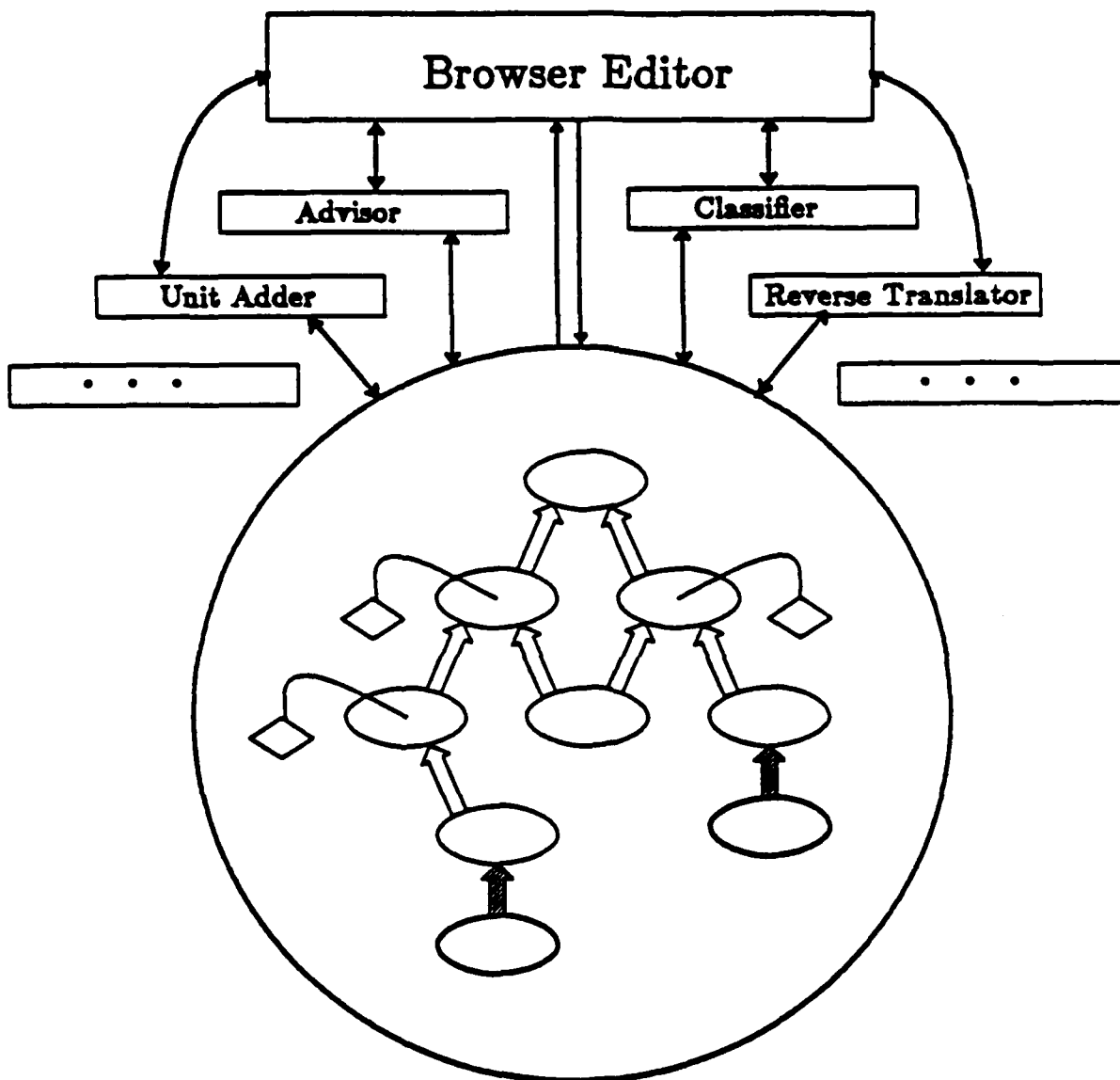
AdaKNET provides the taxonomic structure for a domain. Using the information web defined via AdaKNET, a user can locate components through multiple access paths. Information about components and their relationships needs to be stored only once at the proper level and, through inheritance, that information is available wherever it is needed. This localization of information also applies to rule bases that enable the network to explain itself to the novice or casual user. AdaKNET also is able to support the representation of incompletely specified components including generics and the use of part generators.

AdaTAU's principal role for librarians is to provide navigational advice to the user. Information that is not readily apparent in the network, or information regarding components that are distant (in terms of network links, from one another, can be supplied through attached rule bases. In addition, AdaTAU rules can be provided for safe component addition to the library, as well as part qualification.

Figure 2 provides a skeletal view of the Ada benchmark librarian application. This application assumes an initial browser-style interface so that the user controls how and where to look for information. In addition, other user modes are provided (classifier, advisor, adder, etc.) that cause the application to take a more active stance in support of the user.

Ada Language Issues and Experiences

Many issues relating to Ada and its connection to the design and implementation of reusable systems were discovered during the course of building the RLF system. These issues will be explored further during the continuing work on the RLF. The RLF system itself makes heavy use of generics and relies greatly on the use of dynamic memory to support its storage of network and rule base information. In several cases,



**AdaKNET Model of the Benchmark Domain
with attached AdaTAU Rule Bases**

Figure 2. Librarian Architecture

particular compilation systems did not perform adequately in these areas. In other cases, Ada code could have been made considerably more readable and general if some

Ada restrictions were not in place. One particular realization of this is the lack of a package type in Ada.

Some of the reuse support provided through the RLF could be considerably enhanced by an integration of RLF knowledge bases with the Ada library structure managed by the Ada compilation system used in conjunction with the RLF. This kind of interface is important for tightly integrated reuse-support systems. The RLF system also exposed some needless portability obstacles in relation to source code location (within host computer file systems) and Ada library-based restrictions.

Future Evolution

The RLF provides basic technology that is pointing in the right direction. Crucial features of this technology include:

- the ability to support the matching of reuse techniques to domains (particular techniques include constructive, knowledge-base-assisted and generation);
- a separately maintained domain model; and
- support for domain evolution.

An organon [Simoes88] is the culmination of the RLF technology. From the dictionary, an organon is defined to be "... an instrument for acquiring knowledge; specifically, a body of methodological doctrine comprising principles for scientific and philosophical procedure and investigation". RLF features and capabilities will be enhanced over time to support library content evolution (e.g., replace family of part variants with a suitable generator); automatic maintenance of library content and persistent user models; and, automatic solicitation for new components to cover gaps in library coverage.

An organon will effectively support wide-spectrum reuse including requirements, design and test cases. In the end, an organon is a central repository of domain expertise that effectively combines people, plus emerging and maturing methods, plus supporting technology.

References

- [CAMP85] C. M. Anderson and D. G. McNicholl, "Common Ada Missile Packages (CAMP): Preliminary Technical Report, Vol. 1," *STARS Workshop Proceedings*, April 1985. Contract FO 8635-84-C-0280.
- [Payton82] T. F. Payton, S. E. Keller, J. A. Perkins, S. Rowan, and S. P. Mardinly, "SSAGS: A Syntax and Semantics Analysis and Generation System," *Proceedings of COMPSAC '82*, 1982, pp. 424-433.
- [Prieto-Dias87a] R. Prieto-Dias and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, 4(1) (January 1987), pp. 6-16.
- [Prieto-Dias87b] R. Prieto-Dias, "Domain Analysis for Reusability," *Proceedings of COMPSAC 87*, Tokyo, Japan, October 1987.
- [Simos88] M. Simos, "The Growing of an Organon: A Hybrid Knowledge-Based Technology and Methodology for Software Reuse," *Proceedings of 1988 National Institute for Software Quality and Productivity (NISQP) Conference on Software Reusability*, April 1988, pp. E-1 through E-25.
- [Solderitsch88] J. Solderitsch, M. Simos, and K. Wallnau, "Reusability Library Framework (RLF)," *Conference Proceedings of TRI-Ada '88*, October 1988, pp. 250-257.
- [Solderitsch89] J. Solderitsch, K. Wallnau, and J. Thalhamer, "Constructing Domain-Specific Ada Reuse Libraries," *Proceedings of Seventh Annual National Conference on Ada Technology*, March 1989.
- [Wallnau88] K. Wallnau, J. Solderitsch, M. Simos, R. McDowell, K. Cassell, and D. Campbell, "Construction of Knowledge-Based Components and Applications in Ada," *Proceedings of AIDA-88, Fourth Annual Conference on Artificial Intelligence & Ada*, November 1988, pp. 3-1 through 3-21.



DEPARTMENT OF THE ARMY

US ARMY INFORMATION SYSTEMS SOFTWARE DEVELOPMENT CENTER-WASHINGTON
FORT BELVOIR, VIRGINIA 22060-5456

REPLY TO
ATTENTION OF

APR 12 1989

RAPID Center

James Baldo, Jr.
Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311-1772


Dear Mr. Baldo:

Attached is a position paper written to gain the participation of Terry Vogelsong in the Reuse in Practice Workshop to be hosted jointly by ACM SIGAda, the Institute for Defense Analyses, and the Software Engineering Institute from July 11-13, 1989. Mr. Vogelsong is a technical staff member of the Reusable Ada Packages for Information Systems Development (RAPID) Center within the U.S. Army Information Systems Software Development Center - Washington (SDC-W) located in Falls Church, VA. SDC-W's mission is to develop and maintain management information systems for the Army.

Mr. Vogelsong, as RAPID's representative, has served in the U.S. Army Information Systems Software Center since 1985 and has been a part of the RAPID project since Jan 1989. He has attended several conferences and reuse workshops and has briefed RAPID to industry and internal organizations. His knowledge of RAPID and how to support both developers and users makes him an excellent candidate to attend the Reuse In Practice Workshop. Mr. Vogelsong can be contacted at (703) 756-5202/5003 or at the following address:

USAISSDCW
ATTN: ASQBI-WRC STOP H-4 (Terry Vogelsong)
Fort Belvoir, VA 22060-5456

It is critical that RAPID participate in this workshop to learn how others have resolved reuse issues. As RAPID will be operational on May 1, 1989 and the workshop occurs in July, RAPID may have answers to issues to be discussed. Having a functioning library system makes RAPID a qualified candidate for attendance. RAPID looks forward to receiving an invitation for Terry Vogelsong to participate in the Reuse Workshop to held in Pittsburgh, PA from July 11-13, 1989.


ISAAC H. CHAPPELL, JR.
Colonel, AG
Commanding

REUSABLE ADA PACKAGES FOR INFORMATION SYSTEM DEVELOPMENT (RAPID) -
AN OPERATIONAL CENTER OF EXCELLENCE FOR SOFTWARE REUSE

The following position paper describes the Reusable Ada Packages for Information Systems Development (RAPID) Center located within the U.S. Army Information Systems Software Development Center - Washington (SDC-W) located in Falls Church, VA. SDC-W's mission is to develop and maintain management information systems for the Army. Particular emphasis is placed on the library system used to catalog, analyze, and retrieve reusable software components.

Overview

RAPID, initiated in July 1987, was conceived and developed in support of the Department of Defense Ada initiative. The major role of the RAPID program is to promote the reuse of Ada software, and further, to reduce the cost of system development and maintenance. While providing real services and accomplishing useful tasks during prototype operations, the RAPID Center is a learning laboratory: evaluating and refining reuse methods and techniques; accumulating a store of experience and a cadre of experienced personnel; and refining RAPID guidelines and procedures.

Phase I of RAPID, completed in January 1989, included developing the software for the RAPID Center Library system and authoring initial policy and procedure documents. Phase 2, which begins on May 1, 1989, is an 18-month Pilot RAPID Center operation. During the first nine months, a single development effort will be extensively supported to prove reusability concepts, refine library software, and resolve contractual and management issues. With a limited scale of operation, mistakes and reusability issues can be detected early and corrected or resolved with minimal damage. The remaining nine months will test the feasibility of servicing five Software Development Centers within the U.S. Army Information Systems Software Center (ISSC). Follow-on phases include expanding service to all of the U.S. Army Information Systems Engineering Command (ISEC), Department of the Army, etc., as needed and as funding allows.

After the successful pilot operation is completed, the scope of RAPID will extend to multiple projects, hardware, operating systems, and organizations. The initial domain analysis covered only information management systems, i.e., financial, logistical, tactical management information, communication, personnel/force accounting, and miscellaneous "other software" systems. But the policies, procedures, and guidelines developed in support of RAPID reuse are generic and evolutionary and therefore should apply to any domain.

Staffing

RAPID will have a staff tailored to perform and train Ada reuse, encouraging design methods and architectures that build from reusable components. Ada Consultants or Engineers from RAPID will provide consultation on reuse throughout the entire life cycle of project development and technical assistance on the use of the RAPID Center library and its reusable components. RAPID System Analysts or Designers will attend project reviews to advise on reuse issues, stay abreast of projects, advise project staffs, identify opportunities to reuse existing components, identify potential reusable components, and provide guidance and support to programmers integrating reusable components and documentation into applications.

Command-wide training by RAPID staff on Ada reuse will include the following course topics:

- a. How to implement reuse throughout the life-cycle, from conception through maintenance.
- b. How to write reusable, portable, maintainable components.
- c. How to use the RAPID Center Library, including how to insert components into new development efforts.

RAPID Center Library

The heart of RAPID is the RAPID Center Library (RCL). RCL is much more than a "repository." RCL is an operational, interactive library system used for the identification, analysis, and retrieval of Ada reusable software components. RCL operates on a Digital MicroVAX II located at SDC-W and consists of 30,000 lines of Ada code. The system was designed to be dynamic or modifiable to adjust to the supported domain. Modification is performed through internal system library functions via a menu or keypad keys. None of the 30,000 lines of Ada code need be changed to support a modification.

RCL user functions involve the identification and extraction of a reusable software component (RSC). The user identifies the requirements of a component needed through a faceted classification scheme (explained below). The library takes the description, searches for, and displays a list of "candidate" RSCs. Internal system tools aid in the selection of an RSC. The present tools are:

- a. Analyzing the candidate list of RSCs.
 1. Displaying the number of times each RSC is used,
 2. Displaying the number of reported problems,
 3. Displaying a reusability measure, and
 4. Displaying a complexity measure.

Present RCL tools continued:

- b. Browsing through an individual RSC on candidate list.
 - 1. Viewing the RSC abstract,
 - 2. Viewing the RSC description,
 - 3. Viewing a list of documents that support RSC,
 - 4. Viewing list of problem reports and text of each problem report if necessary, and
 - 5. Viewing numeric measures of lines of code, number of uses, outstanding problems, etc.
- c. Extracting a selected RSC from candidate list or downloading the code and documentation.
- d. Maintaining a search session or "candidate" list of RSCs.
 - 1. Saving the session,
 - 2. Restoring a saved session,
 - 3. Deleting a saved session, or
 - 4. Clearing the session.

Faceted Classification Scheme

RCL uses a faceted classification scheme to store and retrieve RSCs. The classification scheme is a method by which the universe of knowledge is built up or "synthesized" from the elemental classes. Synthesis is the process of assembling elemental classes to express a superimposed, complex, or compound class. Facets are the arranged groups of elemental classes that make up the scheme. This scheme is based upon the Ruben Prieto-Diaz approach of two groups of describers. The first is the "functionality" group consisting of function, objects, and medium attributes. The second is the "environment" group, consisting of system type, functional area, and setting types of facets.

Facets -- properties an RSC may have -- represent different ways of looking at a component. For each facet, specific descriptive terms, called facet terms, classify an RSC within that facet. Terms with the same meaning are known as synonyms, and a group of synonyms is a concept. One term from such a group is selected as the representative term (the "name" of the concept) and serves as the facet term used for the actual classification of RSCs. The remaining synonyms are keyed to the representative term in a list called a thesaurus. When users enter terms to describe the desired component, any one of the synonyms is equivalent to the representative term.

Not every facet need be employed in classifying an RSC. However, the facet's function, language, and certification level should always be given. More than one facet term may be given for a single facet. Component classification can be changed or augmented as required. The RAPID Library classification scheme is designed to allow additions and changes to improve its descriptive power.

The initial set of facets used by RAPID is as follows:

- a. Function - the process the component performs, such as SORT, SIGN, DELETED, etc.
- b. Object - the conceptual object the component operates on, such as STACK, WINDOW, PERSON, etc.
- c. Algorithm - any special method name associated with the function, such as BUBBLE for the function SORT.
- d. Data Representation - the data structure for the physical representation of the object within the component, such as LINKED LIST, RECORD, POINTER, ARRAY, etc.
- e. Unit Type - the program structure of the component, such as FUNCTION, SUBROUTINE, PACKAGE, TASK, etc.
- f. Hardware - the hardware configuration(s) on which the component operates, such as VAX, RATIONAL, etc.
- g. Operating System - operating system associated with the hardware configuration such as VMS, MVS, UNIX, etc.
- h. Language - the programming language that the component is written in, such as ADA, PASCAL, C, etc.
- i. Area of Application - the application area that the component applies to, such as PERSONNEL, LOGISTICS, etc.
- j. Degree of Certification - an indication of the certification level of the component, such as TESTED or CERTIFIED.

RAPID Library System Features

RCL system logs a variety of information for the purpose of tracking its performance, evaluating possible changes to enhance the system, and triggering RAPID Center activities. The logged information includes RSC use, search failures, suggestion box, and user accounts. Some of the data is automatically incorporated into the RSC catalog, while other information is available through reports. These reports and feedback mechanisms aid the RCL System Administrator in determining if the search apparatus needs to be modified or other actions need to be taken. The primary function of the System Administrator is the storage and cataloging of RSCs. Additional duties include maintaining user accounts, logs, suggestions, and updating RSC analysis and search features.

When a user extracts an RSC from the library, a date must be specified when he or she expects to be able to provide "feedback" about successes or problems with the RSCs. These experiences are analyzed and several actions may be taken, as appropriate: updating the RSC's use history in the library, initiating a problem report, recommending enhancements to the RSC, recommending new RSCs, or recommending changes to the library search apparatus. Feedback is solicited about functional fit, cost savings, ease of installation, actual versus expected performance, problems, recommendations for improvement, and any other user comments. Another example of a system generated log is search failure information. This may result in recommendations for new or enhanced RSCs or in updates to RCL search mechanisms (i.e., the classification scheme and the underlying thesaurus), depending on the cause of the search failure.

Reusable Software Components

The RAPID Center Library system is presently operational and the RAPID staff is populating the library with RSCs. Sources of RSCs include reviews of ongoing projects, commercial off the shelf, fielded systems, and RAPID developed. ISEC's reusable goals that pertain to the reusable component are maximum reusability, efficiency, flexibility, ease of use, and protection against misuse.

All RSCs will be evaluated for quality, usefulness, complexity, portability, and profitability. The checklist below will be used to determine if an RSC is to be stored in the library:

- a. Review for portability in accordance with ISEC Portability Guidelines.
- b. Review for reusability in accordance with ISEC Reusability Guidelines.
- c. Review for reliability.
- d. Review for maintainability.
- e. Review for proper testing and test data.
- f. Review for complete documentation - abstract, reuser's manual, function, interfaces, etc.

The evaluation of RSCs will be aided by the use of an automated Ada Measurement and Analysis Tool that measures and evaluates the quality and software factors of reusability, reliability, portability, and maintainability of the developed software. Exactly which tools will be used and to what measurement levels the tools will be set will be determined during pilot operation. The ultimate goal will be to include components that are of high quality, documented, and tested.

Summary

Establishing and operating the RAPID Center required a commitment of resources and personnel by SDC-W management. Until software reuse becomes a "way of life," the RAPID Center must lead the way. Many issues remain to be resolved, several of which will be discussed at the Reuse In Practice Workshop. But with sound policy and attention to the needs and perceptions of the supported development staffs, the RAPID program will more than pay for itself. RAPID is truly a center of excellence for software reuse.

Distribution List for IDA Document D-754

NAME AND ADDRESS	NUMBER OF COPIES
-------------------------	-------------------------

Sponsor

Lt Col James Sweeder SDIO/ENA The Pentagon, Room 1E149 Washington, DC 20301-7100	3
---	----------

Others

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
---	----------

Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1
--	----------

Dr. Keith Bromley Code 7601T Naval Ocean Systems Center 271 Catalina Blvd. San Diego, CA 92152-5000	1
---	----------

Ms. Christine Braun Contel Technology Center 15000 Conference Center Drive P.O. Box 10814 Chantilly, VA 22021-3808	1
--	----------

Brian Baker NAVDAC 8th & M Street, S.E. Washington, Navy Yard Washington, DC 20374	1
--	----------

Jim Perry GTE Govenment Systems 77A Street Building 12 Needham, MA 02194	1
--	----------

NAME AND ADDRESS**NUMBER OF COPIES**

Capt. Jack Rothrock
USAISDCW STOP H-4
Fort Belvoir, VA 22060-5456

1

CSED Review Panel
Dr. Dan Alpert, Director
Program in Science, Technology & Society
University of Illinois
Room 201
912-1/2 West Illinois Street
Urbana, Illinois 61801

1

Dr. Ruth Davis
The Pymatuning Group, Inc.
2000 N. 15th Street, Suite 707
Arlington, Va 22201

1

Dr. Thomas C. Brandt
10302 Bluet Terrace
Upper Marlboro, MD 20772

1

Dr. C.E. Hutchinson, Dean
Thayer School of Engineering
Dartmouth College
Hanover, NH 03755

1

Mr. A.J. Jordano
IBM, Federal Systems Division
6600 Rockledge Dr.
Bethesda, MD 20817

1

Dr. John M. Palms, President
Georgia State University
President's Office
University Plaza
Atlanta, GA 30303

1

Dr. Ernest W. Kent
Philips Laboratories
345 Scarborough Road
Briarcliff Manor, NY 10510

1

NAME AND ADDRESS**NUMBER OF COPIES**

Mr. Keith Uncapher, Associate Dean
School of Engineering
University of Southern California
Olin Hall
330A University Park
Los Angeles, CA 90089-1454

1

IDA

General W.Y., Smith, HQ

1

Mr. Philip L. Major, HQ

1

Dr. Robert E. Roberts, HQ

1

Ms. Ruth L. Greenstein, HQ

1

Ms. Anne Douville, CSED

1

Mr. Terry Mayfield, CSED

1

Dr. Richard Ivanetich, CSED

1

Dr. Richard Wexelblat, CSED

1

Dr. Dennis Fife, CSED

1

Mr. James Baldo, CSED

2

Mr. David Wheeler, CSED

1

Dr. Norman Howes, CSED

1

Mr. Steve Edwards, CSED

1

Dr. Craig Will, CSED

1

Mr. Robert Knapper, CSED

1

Mr. Michael Bloom, CSED

1

Ms. Beth Springsteen, CSED

1

Ms. Sylvia Reynolds, CSED

2

IDA Control & Distribution Vault

3